# Oracle® Call Interface

## Programmer's Guide

19c

E96204-01

January 2019

Oracle Call Interface Programmer's Guide, 19c

E96204-01

Copyright © 1996, 2019, Oracle and/or its affiliates. All rights reserved.

Primary Author: Mamata Basapur

Contributing Authors: Rod Ward

Contributors: D. Adams, G. Arora, V. Arora, A. Bande, D. Banerjee, S. Banerjee, M. Bastawala, E. Belden, P. Betteridge, N. Bhatt, T. Bhosle, J. Blowney, R. Chakravarthula, S. Chandrasekar, B. Cheng, D. Chiba, L. Chidambaran, C. Colrain, T. Das, Ronald Decker, A. Desai, A. Downing, S. Fogel, T. Hoang, N. Ikeda, K. Itikarlapalli, C. Iyer, S. Iyer, V. Jitta, C. Jones, A. Keh, B. Khaladkar, S. Krishnaswamy, R. Kumar, R. Kumar, S. Lahorani, S. Lari, T. H. Lee, T. Li, C. Liang, I. Listvinsky, J. Liu, E. Lu, S. Lynn, K.Mensah, V. Moore, A. Mullick, K. Neel, M. Orgiyan, E. Paapanen, S. Pelski, R. Phillips, R. Pingte, R. Rajamani, M. Ramacher, A. Ramappa, S. Sahu, A. Saxena, S. Seshadri, R. Singh, B. Sinha, H. Slattery, J. Stewart, L. Sun, S. Suresh, S. Tata, H. Tran, A. Tuininga, S. Vallapureddy, M. Vemana, S. Vemuri, B. Venkatakrishnan, K. Verma, G. Viswanathan, S. Wolicki, L. Wong, S. Youssef, B. Zebian

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

# Contents

## Preface

Audience lviii
Documentation Accessibility lviii
Related Documents lix
Conventions lx

## Changes in This Release for Oracle Call Interface Programmer's Guide

Changes in Oracle Database Release 19c Version 19.1 lxi
Changes in Oracle Database Release 18c Version 18.1 lxi

## 1 OCI: Introduction

1.1 Overview of OCI 1-1
1.2 Building an OCI Application 1-3
1.3 Alternatives to OCI 1-3
1.4 SQL Statements 1-5
  1.4.1 Data Definition Language 1-5
  1.4.2 Control Statements 1-6
  1.4.3 Data Manipulation Language 1-6
  1.4.4 Queries 1-7
  1.4.5 PL/SQL 1-7
  1.4.6 Embedded SQL 1-8
  1.4.7 Special OCI Terms for SQL 1-9
1.5 Procedural and Nonprocedural Elements 1-9
1.6 Object Support 1-10
  1.6.1 Client-Side Object Cache 1-11
  1.6.2 Associative and Navigational Interfaces 1-11
  1.6.3 OCI Runtime Environment for Objects 1-12
  1.6.4 Type Management: Mapping and Manipulation Functions 1-12
  1.6.5 Object Type Translator 1-13
1.7 Simple Oracle Document Access (SODA) 1-13

1.8 Encapsulated Interfaces 1-14
1.9 User Authentication and Password Management 1-14
1.10 Features to Improve Application Performance and Scalability 1-15
1.11 Oracle Database Advanced Queuing 1-15
1.12 XA Library Support 1-16

# 2 Oracle Instant Client and Oracle Instant Client Light

2.1 About Oracle Instant Client 2-1
2.2 Install from ZIP Files, from RPMs, and from Oracle Universal Installer 2-3
2.3 Environment Variables for Oracle Instant Client 2-5
2.4 Database Connection Strings for Oracle Instant Client 2-6
2.5 SDK for Oracle Instant Client 2-7
2.6 About Oracle Instant Client Light 2-7
  2.6.1 Globalization Settings 2-8
  2.6.2 Libraries for Oracle Instant Client Light 2-9
  2.6.3 Installing Oracle Instant Client Light 2-10
2.7 About Patching Oracle Instant Client Shared Libraries on Linux or UNIX 2-11
2.8 Regeneration of Data Shared Library and Zip and RPM Files 2-11
  2.8.1 Regenerating Data Shared Library libociei.so 2-12
  2.8.2 Regenerating Data Shared Library libociicus.so 2-12
  2.8.3 Regenerating Data Shared Libraries libociei.so and libociicus.so in One Step 2-13
  2.8.4 Regenerating Zip and RPM Files for the Basic Package 2-13
  2.8.5 Regenerating Zip and RPM Files for the Basic Light Package 2-13
  2.8.6 Regenerating Zip and RPM Files for the JDBC Package 2-13
  2.8.7 Regenerating Zip and RPM Files for the ODBC Package 2-13
  2.8.8 Regenerating Zip and RPM Files for the SQL*Plus Package 2-14
  2.8.9 Regenerating Zip and RPM Files for the Tools Package 2-14
  2.8.10 Regenerating Zip and RPM Files for All Packages 2-14

# 3 Building and Configuring OCI Applications

3.1 Header File and Makefile Locations 3-1
3.2 Building an OCI Application on Linux and UNIX 3-2
  3.2.1 Oracle Directory Structure 3-2
  3.2.2 Demonstration OCI Programs 3-2
3.3 Building an Application on Windows 3-3
3.4 Database Connection Strings 3-4
  3.4.1 Examples of Oracle Database Connection String Connect Identifiers 3-5
3.5 Client and Server Operating with Different Versions of Time Zone Files 3-6
3.6 OCI Client-Side Deployment Parameters Using oraaccess.xml 3-7

3.6.1 About oraaccess.xml 3-7
3.6.2 About Client-Side Deployment Parameters Specified in oraaccess.xml 3-8
3.6.3 High Level Structure of oraaccess.xml 3-8
3.6.4 About Specifying Global Parameters in oraaccess.xml 3-9
3.6.5 About Specifying Defaults for Connection Parameters 3-11
3.6.6 Overriding Connection Parameters at the Connection-String Level 3-14
3.6.7 About OCI Session Pool Configuration in oraaccess.xml 3-17
3.6.8 File (oraaccess.xml) Properties 3-19
3.7 About Compatibility and Upgrading 3-20
3.7.1 Oracle Client and Server Cross Version Compatibility 3-20
3.7.2 Version Compatibility of Statically Linked and Dynamically Linked Applications 3-20
3.7.3 About Upgrading of Existing OCI Release 7 Applications 3-21
3.7.4 Obsolete OCI Routines 3-22
3.7.5 OCI Routines Not Supported 3-24
3.8 Fault Diagnosability in OCI 3-25
3.8.1 About Fault Diagnosability in OCI 3-25
3.8.2 ADR Base Location 3-25
3.8.3 Using ADRCI 3-27
3.8.4 Controlling ADR Creation and Disabling Fault Diagnosability Using sqlnet.ora 3-29

## 4 OCI Programming Basics

4.1 Overview of OCI Program Programming 4-1
4.2 OCI Data Structures 4-3
4.2.1 Handles 4-3
4.2.1.1 About Allocating and Freeing Handles 4-5
4.2.1.2 Environment Handle 4-6
4.2.1.3 Error Handle 4-6
4.2.1.4 Service Context Handle and Associated Handles 4-6
4.2.1.5 Statement, Bind, and Define Handles 4-8
4.2.1.6 Describe Handle 4-9
4.2.1.7 Complex Object Retrieval Handle 4-9
4.2.1.8 Thread Handle 4-10
4.2.1.9 Subscription Handle 4-10
4.2.1.10 Direct Path Handles 4-10
4.2.1.11 Connection Pool Handle 4-11
4.2.1.12 Handle Attributes 4-11
4.2.1.13 OCI Descriptors 4-12
4.3 OCI Programming Steps 4-17
4.3.1 OCI Environment Initialization 4-18

4.3.1.1 About Creating the OCI Environment 4-18
4.3.1.2 About Allocating Handles and Descriptors 4-19
4.3.1.3 Application Initialization, Connection, and Session Creation 4-19
4.3.2 About Processing SQL Statements in OCI 4-25
4.3.3 Commit or Roll Back Operations 4-25
4.3.4 About Terminating the Application 4-26
4.3.5 Error Handling in OCI 4-27
4.3.5.1 Return and Error Codes for Data 4-28
4.3.5.2 Functions Returning Other Values 4-29
4.4 Additional Coding Guidelines 4-29
4.4.1 Operating System Considerations 4-29
4.4.2 Parameter Types 4-30
4.4.2.1 Address Parameters 4-30
4.4.2.2 Integer Parameters 4-30
4.4.2.3 Character String Parameters 4-31
4.4.3 Inserting Nulls into a Column 4-31
4.4.4 Indicator Variables 4-31
4.4.4.1 Input 4-32
4.4.4.2 Output 4-32
4.4.4.3 Indicator Variables for Named Data Types and REFs 4-32
4.4.5 About Canceling Calls 4-33
4.4.6 Positioned Updates and Deletes 4-34
4.4.7 Reserved Words 4-35
4.4.7.1 Oracle Reserved Namespaces 4-35
4.4.8 Polling Mode Operations in OCI 4-36
4.4.9 Nonblocking Mode in OCI 4-36
4.4.10 Setting Blocking Modes 4-37
4.4.11 Canceling a Nonblocking Call 4-38
4.5 About Using PL/SQL in an OCI Program 4-38
4.6 OCI Globalization Support 4-39
4.6.1 Client Character Set Control from OCI 4-40
4.6.2 Character Control and OCI Interfaces 4-40
4.6.3 Character-Length Semantics in OCI 4-41
4.6.4 Character Set Support in OCI 4-41
4.6.5 Controlling Language and Territory in OCI 4-41
4.6.6 Other OCI Globalization Support Functions 4-42
4.6.7 About Getting Locale Information in OCI 4-42
4.6.8 About OCI and the BOM (Byte Order Mark) 4-43
4.6.9 About Manipulating Strings in OCI 4-44
4.6.10 About Converting Character Sets in OCI 4-45
4.6.11 OCI Messaging Functions 4-46

4.6.12 Imsgen Utility 4-47
    4.6.12.1 Guidelines for Text Message Files 4-47
    4.6.12.2 An Example of Creating a Binary Message File from a Text Message File 4-48

## 5 Data Types

5.1 Oracle Data Types 5-1
  5.1.1 About Using External Data Type Codes 5-3
5.2 Internal Data Types 5-4
  5.2.1 LONG, RAW, LONG RAW, VARCHAR2 5-5
  5.2.2 Character Strings and Byte Arrays 5-5
  5.2.3 UROWID 5-6
  5.2.4 BINARY_FLOAT and BINARY_DOUBLE 5-7
5.3 External Data Types 5-7
  5.3.1 VARCHAR2 5-10
    5.3.1.1 Input 5-10
    5.3.1.2 Output 5-11
  5.3.2 NUMBER 5-12
  5.3.3 64-Bit Integer Host Data Type 5-13
    5.3.3.1 OCI Bind and Define for 64-Bit Integers 5-13
    5.3.3.2 Support for OUT Bind DML Returning Statements 5-14
  5.3.4 INTEGER 5-15
  5.3.5 FLOAT 5-15
  5.3.6 STRING 5-15
    5.3.6.1 Input 5-15
    5.3.6.2 Output 5-16
  5.3.7 VARNUM 5-16
  5.3.8 LONG 5-17
  5.3.9 VARCHAR 5-17
  5.3.10 DATE 5-17
  5.3.11 RAW 5-18
  5.3.12 VARRAW 5-19
  5.3.13 LONG RAW 5-19
  5.3.14 UNSIGNED 5-19
  5.3.15 LONG VARCHAR 5-19
  5.3.16 LONG VARRAW 5-19
  5.3.17 CHAR 5-20
    5.3.17.1 Input 5-20
    5.3.17.2 Output 5-21
  5.3.18 CHARZ 5-21
  5.3.19 Named Data Types: Object, VARRAY, Nested Table 5-22

5.3.20 REF 5-22
5.3.21 ROWID Descriptor 5-23
5.3.22 LOB Descriptor 5-23
5.3.22.1 BFILE 5-25
5.3.22.2 BLOB 5-26
5.3.22.3 CLOB 5-26
5.3.22.4 NCLOB 5-26
5.3.23 Datetime and Interval Data Type Descriptors 5-26
5.3.23.1 ANSI DATE 5-27
5.3.23.2 TIMESTAMP 5-27
5.3.23.3 TIMESTAMP WITH TIME ZONE 5-27
5.3.23.4 TIMESTAMP WITH LOCAL TIME ZONE 5-28
5.3.23.5 INTERVAL YEAR TO MONTH 5-28
5.3.23.6 INTERVAL DAY TO SECOND 5-28
5.3.23.7 About Avoiding Unexpected Results Using Datetime 5-28
5.3.24 Native Float and Native Double 5-29
5.3.25 C Object-Relational Data Type Mappings 5-29
5.4 Data Conversions 5-29
5.4.1 Data Conversions for LOB Data Type Descriptors 5-31
5.4.2 Data Conversions for Datetime and Interval Data Types 5-31
5.4.2.1 Assignment Notes 5-32
5.4.2.2 Data Conversion Notes for Datetime and Interval Types 5-33
5.4.3 Datetime and Date Upgrading Rules 5-33
5.4.3.1 Pre-9.0 Client with 9.0 or Later Server 5-33
5.4.3.2 Pre-9.0 Server with 9.0 or Later Client 5-33
5.4.4 Data Conversion for BINARY_FLOAT and BINARY_DOUBLE in OCI 5-34
5.5 Typecodes 5-35
5.5.1 Relationship Between SQLT and OCI_TYPECODE Values 5-36
5.6 Definitions in oratypes.h 5-38

## 6 Using SQL Statements in OCI

6.1 Overview of SQL Statement Processing 6-1
6.2 About Preparing Statements 6-4
6.2.1 About Using Prepared Statements on Multiple Servers 6-5
6.3 About Binding Placeholders in OCI 6-6
6.3.1 Rules for Placeholders 6-7
6.4 About Executing Statements 6-7
6.4.1 Execution Snapshots 6-8
6.4.2 Execution Modes of OCIStmtExecute() 6-9
6.4.2.1 Using Batch Error Mode 6-9

6.4.2.2 Example of Batch Error Mode 6-11
6.5 About Describing Select-List Items 6-12
6.5.1 Implicit Describe 6-13
6.5.2 Explicit Describe of Queries 6-15
6.6 About Defining Output Variables in OCI 6-16
6.7 About Fetching Results 6-16
6.7.1 About Fetching LOB Data 6-17
6.7.2 About Setting Prefetch Count 6-17
6.8 About Using Scrollable Cursors in OCI 6-19
6.8.1 About Increasing Scrollable Cursor Performance 6-20
6.8.2 Example of Access on a Scrollable Cursor 6-21

## 7 Binding and Defining in OCI

7.1 Overview of Binding in OCI 7-1
7.1.1 Named Binds and Positional Binds 7-3
7.1.2 OCI Array Interface 7-4
7.1.3 About Binding Placeholders in PL/SQL 7-5
7.1.4 Steps Used in OCI Binding 7-6
7.1.5 PL/SQL Block in an OCI Program 7-7
7.2 Advanced Bind Operations in OCI 7-9
7.2.1 About Binding LOBs 7-10
7.2.1.1 Binding LOB Locators 7-10
7.2.1.2 About Binding LOB Data 7-12
7.2.2 About Binding in OCI_DATA_AT_EXEC Mode 7-16
7.2.3 About Binding REF CURSOR Variables 7-17
7.3 Overview of Defining in OCI 7-17
7.3.1 Steps Used in OCI Defining 7-18
7.4 Advanced Define Operations in OCI 7-19
7.4.1 About Defining LOB Output Variables 7-20
7.4.1.1 About Defining LOB Locators 7-20
7.4.1.2 About Defining LOB Data 7-21
7.4.2 About Defining PL/SQL Output Variables 7-22
7.4.3 About Defining for a Piecewise Fetch 7-22
7.5 About Binding and Defining Arrays of Structures in OCI 7-22
7.5.1 Skip Parameters 7-23
7.5.1.1 Skip Parameters for Standard Arrays 7-24
7.5.2 OCI Calls Used with Arrays of Structures 7-24
7.5.3 Arrays of Structures and Indicator Variables 7-25
7.6 About Binding and Defining Multiple Buffers 7-25
7.7 DML with a RETURNING Clause in OCI 7-29

7.7.1 About Using DML with a RETURNING Clause to Combine Two SQL Statements 7-29
7.7.2 About Binding RETURNING...INTO Variables 7-30
7.7.3 OCI Error Handling 7-31
7.7.4 DML with RETURNING REF...INTO Clause in OCI 7-31
7.7.4.1 Binding the Output Variable 7-31
7.7.5 Additional Notes About OCI Callbacks 7-33
7.7.6 Array Interface for DML RETURNING Statements in OCI 7-33
7.8 Character Conversion in OCI Binding and Defining 7-33
7.8.1 About Choosing a Character Set 7-34
7.8.1.1 Character Set Form and ID 7-35
7.8.1.2 Implicit Conversion Between CHAR and NCHAR 7-35
7.8.2 About Setting Client Character Sets in OCI 7-36
7.8.3 About Binding Variables in OCI 7-36
7.8.3.1 About Using the OCI_ATTR_MAXDATA_SIZE Attribute 7-37
7.8.3.2 About Using the OCI_ATTR_MAXCHAR_SIZE Attribute 7-38
7.8.3.3 Buffer Expansion During OCI Binding 7-38
7.8.3.4 Constraint Checking During Defining 7-39
7.8.3.5 General Compatibility Issues for Character-Length Semantics in OCI 7-40
7.9 PL/SQL REF CURSORs and Nested Tables in OCI 7-43
7.10 Natively Describe and Bind All PL/SQL Types Including Package Types 7-44
7.11 Runtime Data Allocation and Piecewise Operations in OCI 7-45
7.11.1 Valid Data Types for Piecewise Operations 7-46
7.11.2 Types of Piecewise Operations 7-46
7.11.3 About Providing INSERT or UPDATE Data at Runtime 7-47
7.11.3.1 Performing a Piecewise Insert or Update 7-48
7.11.4 Piecewise Operations with PL/SQL 7-50
7.11.5 PL/SQL Indexed Table Binding Support 7-50
7.11.5.1 Restrictions for PL/SQL Indexed Table Binding Interface 7-51
7.11.6 About Providing FETCH Information at Run Time 7-52
7.11.6.1 Performing a Piecewise Fetch 7-52
7.11.7 Piecewise Binds and Defines for LOBs 7-54

## 8 Describing Schema Metadata

8.1 About Using OCIDescribeAny() 8-1
8.1.1 Limitations on OCIDescribeAny() 8-3
8.1.2 Notes on Types and Attributes 8-3
8.1.2.1 Data Type Codes 8-3
8.1.2.2 About Describing Types 8-4
8.1.2.3 Implicit and Explicit Describe Operations 8-4

8.1.2.4 OCI_ATTR_LIST_ARGUMENTS Attribute 8-5
8.2 Parameter Attributes 8-5
8.2.1 Table or View Parameters 8-7
8.2.2 Procedure, Function, and Subprogram Attributes 8-8
8.2.3 Package Attributes 8-9
8.2.4 Type Attributes 8-9
8.2.5 Type Attribute Attributes 8-11
8.2.6 Type Method Attributes 8-12
8.2.7 Collection Attributes 8-13
8.2.8 Synonym Attributes 8-14
8.2.9 Sequence Attributes 8-15
8.2.10 Column Attributes 8-16
8.2.11 Argument and Result Attributes 8-20
8.2.12 List Attributes 8-22
8.2.13 Schema Attributes 8-22
8.2.14 Database Attributes 8-23
8.2.15 Rule Attributes 8-24
8.2.16 Rule Set Attributes 8-24
8.2.17 Evaluation Context Attributes 8-24
8.2.18 Table Alias Attributes 8-25
8.2.19 Variable Type Attributes 8-25
8.2.20 Name Value Attributes 8-26
8.3 Character-Length Semantics Support in Describe Operations 8-26
8.3.1 Implicit Describing 8-27
8.3.2 Explicit Describing 8-27
8.3.2.1 Client and Server Compatibility Issues for Describing 8-28
8.4 Examples Using OCIDescribeAny() 8-28
8.4.1 Retrieving Column Data Types for a Table 8-28
8.4.2 Describing the Stored Procedure 8-30
8.4.3 Retrieving Attributes of an Object Type 8-31
8.4.4 Retrieving the Collection Element's Data Type of a Named Collection Type 8-33
8.4.5 Describing with Character-Length Semantics 8-35
8.4.6 Describing Each Column to Know Whether It Is an Invisible Column 8-36

## 9 LOB and BFILE Operations

9.1 About Using OCI Functions for LOBs 9-1
9.2 About Creating and Modifying Persistent LOBs 9-2
9.3 About Associating a BFILE in a Table with an Operating System File 9-3
9.4 LOB Attributes of an Object 9-3
9.4.1 Writing to a LOB Attribute of an Object 9-4

9.4.2 Transient Objects with LOB Attributes 9-4
9.5 Array Interface for LOBs 9-5
9.6 About Using LOBs of Size Greater than 4 GB 9-5
9.6.1 Functions to Use for the Increased LOB Sizes 9-6
9.6.2 Compatibility and Migration 9-7
9.7 LOB and BFILE Functions in OCI 9-9
9.7.1 About Improving LOB Read/Write Performance 9-10
9.7.1.1 About Using Data Interface for LOBs 9-10
9.7.1.2 About Using OCILobGetChunkSize() 9-11
9.7.1.3 About Using OCILobWriteAppend2() 9-11
9.7.1.4 About Using OCILobArrayRead() and OCILobArrayWrite() 9-12
9.7.2 LOB Buffering Functions 9-12
9.7.3 Functions for Opening and Closing LOBs 9-13
9.7.3.1 Restrictions on Opening and Closing LOBs 9-14
9.7.4 LOB Read and Write Callbacks 9-15
9.7.4.1 Callback Interface for Streaming 9-15
9.7.4.2 Reading LOBs by Using Callbacks 9-15
9.7.4.3 Writing LOBs by Using Callbacks 9-17
9.8 Temporary LOB Support 9-18
9.8.1 Creating and Freeing Temporary LOBs 9-19
9.8.2 Temporary LOB Durations 9-20
9.8.3 About Freeing Temporary LOBs 9-21
9.8.4 Take Care When Assigning Pointers 9-21
9.8.5 Temporary LOB Example 9-22
9.9 Prefetching of LOB Data, Length, and Chunk Size 9-25
9.10 Options of SecureFiles LOBs 9-28

## 10 Managing Scalable Platforms

10.1 OCI Support for Transactions 10-1
10.2 Levels of Transactional Complexity 10-2
10.2.1 Simple Local Transactions 10-2
10.2.2 Serializable or Read-Only Local Transactions 10-3
10.2.3 Global Transactions 10-3
10.2.3.1 Transaction Identifiers 10-3
10.2.3.2 Attribute OCI_ATTR_TRANS_NAME 10-4
10.2.3.3 Transaction Branches 10-4
10.2.3.4 Branch States 10-6
10.2.3.5 Detaching and Resuming Branches 10-6
10.2.3.6 About Setting the Client Database Name 10-6
10.2.3.7 One-Phase Commit Versus Two-Phase Commit 10-7

10.2.3.8 Preparing Multiple Branches in a Single Message 10-8
10.2.4 Transaction Examples 10-8
10.2.5 Initialization Parameters 10-8
10.2.5.1 Showing Update Successfully, One-Phase Commit 10-9
10.2.5.2 Showing Starting a Transaction, Detach, Resume, Prepare, Two-Phase Commit 10-9
10.2.5.3 Showing a Read-Only Update Fails 10-10
10.2.5.4 Showing Starting a Read-Only Transaction, Select, and Commit 10-11
10.3 Password and Session Management 10-11
10.3.1 OCI Authentication Management 10-11
10.3.2 OCI Password Management 10-13
10.3.3 Secure External Password Store 10-13
10.3.4 OCI Session Management 10-14
10.4 Middle-Tier Applications in OCI 10-15
10.4.1 OCI Attributes for Middle-Tier Applications 10-16
10.4.1.1 OCI_CRED_PROXY 10-16
10.4.1.2 OCI_ATTR_PROXY_CREDENTIALS 10-17
10.4.1.3 OCI_ATTR_DISTINGUISHED_NAME 10-17
10.4.1.4 OCI_ATTR_CERTIFICATE 10-17
10.4.1.5 OCI_ATTR_INITIAL_CLIENT_ROLES 10-18
10.4.1.6 OCI_ATTR_CLIENT_IDENTIFIER 10-18
10.4.1.7 OCI_ATTR_PASSWORD 10-19
10.5 Externally Initialized Context in OCI 10-22
10.5.1 Externally Initialized Context Attributes in OCI 10-22
10.5.1.1 OCI_ATTR_APPCTX_SIZE 10-23
10.5.1.2 OCI_ATTR_APPCTX_LIST 10-23
10.5.1.3 Session Handle Attributes Used to Set an Externally Initialized Context 10-24
10.5.2 End-to-End Application Tracing 10-24
10.5.2.1 OCI_ATTR_COLLECT_CALL_TIME 10-25
10.5.2.2 OCI_ATTR_CALL_TIME 10-25
10.5.2.3 Attributes for End-to-End Application Tracing 10-25
10.5.3 Using OCISessionBegin() with an Externally Initialized Context 10-26
10.6 Client Application Context 10-28
10.6.1 Using Multiple SET Operations 10-29
10.6.2 Using CLEAR-ALL Operations Between SET Operations 10-30
10.6.3 Network Transport and PL/SQL on Client Namespace 10-30
10.7 Using Edition-Based Redefinition 10-31
10.8 OCI Security Enhancements 10-32
10.8.1 Controlling the Database Version Banner Displayed 10-33
10.8.2 Banners for Unauthorized Access and User Actions Auditing 10-33
10.8.3 Non-Deferred Linkage 10-34

10.9 Overview of OCI Multithreaded Development 10-34
  10.9.1 Advantages of OCI Thread Safety 10-35
  10.9.2 OCI Thread Safety and Three-Tier Architectures 10-35
  10.9.3 About Implementing Thread Safety 10-36
    10.9.3.1 About Polling Mode Operations and Thread Safety 10-37
    10.9.3.2 Mixing 7.x and Later Release OCI Calls 10-37
10.10 OCIThread Package 10-37
  10.10.1 Initialization and Termination 10-39
    10.10.1.1 OCIThread Context 10-39
  10.10.2 Passive Threading Primitives 10-40
    10.10.2.1 OCIThreadMutex 10-41
    10.10.2.2 OCIThreadKey 10-41
    10.10.2.3 OCIThreadKeyDestFunc 10-42
    10.10.2.4 OCIThreadId 10-42
  10.10.3 Active Threading Primitives 10-43
    10.10.3.1 OCIThreadHandle 10-43

## 11 Session Pooling and Connection Pooling in OCI

11.1 Session Pooling in OCI 11-1
  11.1.1 Functionality of OCI Session Pooling 11-2
  11.1.2 Homogeneous and Heterogeneous Session Pools 11-2
  11.1.3 About Using Tags in Session Pools 11-2
    11.1.3.1 Multi-Property Tags 11-3
  11.1.4 OCI Handles for Session Pooling 11-6
    11.1.4.1 OCISPool 11-6
    11.1.4.2 OCIAuthInfo 11-6
  11.1.5 Using OCI Session Pooling 11-7
  11.1.6 OCI Calls for Session Pooling 11-8
    11.1.6.1 Allocate the Pool Handle 11-9
    11.1.6.2 Create the Pool Session 11-9
    11.1.6.3 Log On to the Database 11-9
    11.1.6.4 Log Off from the Database 11-10
    11.1.6.5 Destroy the Session Pool 11-11
    11.1.6.6 Free the Pool Handle 11-11
  11.1.7 Example of OCI Session Pooling 11-11
11.2 Database Resident Connection Pooling 11-11
11.3 About Using Oracle Connection Manager in Traffic Director Mode 11-12
11.4 Connection Pooling in OCI 11-15
  11.4.1 OCI Connection Pooling Concepts 11-15
    11.4.1.1 Similarities and Differences from a Shared Server 11-15

11.4.1.2 Stateless Sessions Versus Stateful Sessions 11-16
11.4.1.3 Multiple Connection Pools 11-16
11.4.1.4 Transparent Application Failover 11-17
11.4.2 Using OCI Calls for Connection Pooling 11-17
11.4.2.1 Allocate the Pool Handle 11-18
11.4.2.2 Create the Connection Pool 11-18
11.4.2.3 Log On to the Database 11-19
11.4.2.4 Deal with SGA Limitations in Connection Pooling 11-21
11.4.2.5 Log Off from the Database 11-22
11.4.2.6 Destroy the Connection Pool 11-22
11.4.2.7 Free the Pool Handle 11-23
11.4.3 Examples of OCI Connection Pooling 11-23
11.5 When to Use Connection Pooling, Session Pooling, or Neither 11-23
11.5.1 Functions for Session Creation 11-24
11.5.2 About Choosing Between Different Types of OCI Sessions 11-25

## 12 High Availability in OCI

12.1 Runtime Connection Load Balancing 12-1
12.2 HA Event Notification 12-2
12.2.1 OCIEvent Handle 12-3
12.2.2 OCI Failover for Connection and Session Pools 12-3
12.2.3 OCI Failover for Independent Connections 12-4
12.2.4 Event Callback 12-4
12.2.5 Custom Pooling: Tagged Server Handles 12-5
12.2.6 About Determining Transparent Application Failover (TAF) Capabilities 12-6
12.3 Transparent Application Failover in OCI 12-7
12.3.1 About Configuring Transparent Application Failover 12-8
12.3.2 Transparent Application Failover Callbacks in OCI 12-8
12.3.3 Transparent Application Failover Callback Structure and Parameters 12-9
12.3.4 Failover Callback Structure and Parameters 12-10
12.3.5 Failover Callback Registration 12-11
12.3.6 Failover Callback Example 12-11
12.3.7 Handling OCI_FO_ERROR 12-13
12.4 OCI and Transaction Guard 12-14
12.4.1 Developing Applications that Use Transaction Guard 12-15
12.4.1.1 Typical Transaction Guard Usage 12-16
12.4.1.2 Transaction Guard Examples 12-17
12.5 OCI and Application Continuity 12-22
12.5.1 About Added Support for Application Continuity 12-23
12.5.2 What Happens Following a Recoverable Error 12-23

12.5.3 Criteria for Successful Replay 12-24
12.5.4 What Factors Disable Application Continuity in OCI 12-24
12.5.5 Failed Replay 12-25
12.5.6 When Is Application Continuity Most Effective 12-25
12.5.6.1 When Application Continuity in OCI Can Fail Over 12-26
12.5.6.2 Application Continuity in OCI Does Not Support These Constructs 12-27
12.5.6.3 Possible Side Effects of Application Continuity 12-27

## 13 Notification Methods and Database Advanced Queuing

13.1 About Continuous Query Notification 13-1
13.2 Publish-Subscribe Notification in OCI 13-2
13.2.1 Publish-Subscribe Registration Functions in OCI 13-4
13.2.1.1 Publish-Subscribe Register Directly to the Database 13-4
13.2.1.2 Open Registration for Publish-Subscribe 13-8
13.2.1.3 Using OCI to Open Register with LDAP 13-9
13.2.1.4 Setting QOS, Timeout Interval, Namespace, Client Address, and Port Number 13-10
13.2.1.5 OCI Functions Used to Manage Publish-Subscribe Notification 13-11
13.2.2 Notification Callback in OCI 13-12
13.2.3 Notification Procedure 13-15
13.2.4 Publish-Subscribe Direct Registration Example 13-15
13.2.5 Publish-Subscribe LDAP Registration Example 13-20
13.3 OCI and Database Advanced Queuing 13-22
13.3.1 OCI Database Advanced Queuing Functions 13-23
13.3.2 OCI Database Advanced Queuing Descriptors 13-24
13.3.3 Database Advanced Queuing in OCI Versus PL/SQL 13-25
13.3.4 Using Buffered Messaging 13-30

## 14 User-Defined Callback Functions in OCI

14.1 About Registering User Callbacks in OCI 14-1
14.1.1 OCIUserCallbackRegister 14-3
14.1.2 User Callback Function 14-3
14.1.3 User Callback Control Flow 14-4
14.1.4 User Callback for OCIErrorGet() 14-5
14.1.5 Errors from Entry Callbacks 14-6
14.1.6 Dynamic Callback Registrations 14-6
14.1.7 About Loading Multiple Packages 14-6
14.1.8 Package Format 14-7
14.1.9 User Callback Chaining 14-8

14.1.10 About Accessing Other Data Sources Through OCI 14-8
14.1.11 Restrictions on Callback Functions 14-8
14.1.12 Example of OCI Callbacks 14-9
14.2 OCI Callbacks from External Procedures 14-11

## 15 Performance Topics

15.1 Statement Caching in OCI 15-1
15.1.1 Statement Caching Without Session Pooling in OCI 15-2
15.1.2 Statement Caching with Session Pooling in OCI 15-2
15.1.3 Rules for Statement Caching in OCI 15-3
15.1.4 Bind and Define Optimization in Statement Caching 15-5
15.1.5 OCI Statement Caching Code Example 15-6
15.2 Implicit Fetching of ROWIDs 15-6
15.2.1 About Implicit Fetching of ROWIDs 15-6
15.2.2 Example of Implicit Fetching of ROWIDs 15-8
15.3 OCI Support for Implicit Results 15-9
15.4 Client Result Cache 15-12
15.5 Client Statement Cache Auto-Tuning 15-12
15.5.1 About Auto-Tuning Client Statement Cache 15-12
15.5.2 Benefit of Auto-Tuning Client Statement Cache 15-13
15.5.3 Client Statement Cache Auto-Tuning Parameters 15-13
15.5.3.1 <statement_cache> 15-14
15.5.3.2 <auto_tune> 15-14
15.5.3.3 Comparison of the Connection Specific Auto-Tuning Parameters 15-16
15.5.4 Usage Examples of Client Statement Cache Auto Tuning 15-18
15.5.5 Enabling and Disabling OCI Client Auto-Tuning 15-19
15.5.6 Usage Guidelines for Auto-Tuning Client Statement Cache 15-19

## 16 Database Startup and Shutdown

16.1 About OCI Database Startup and Shutdown 16-1
16.2 Examples of Startup and Shutdown in OCI 16-2

## 17 Support for Pluggable Databases

17.1 Enhancements on OCI API Calls with Multitenant Container Databases (CDB) in General 17-1
17.2 OCI Enhancements for ALTER SESSION SET CONTAINER 17-2
17.3 Restrictions on OCI API Calls with Multitenant Container Databases (CDB) in General 17-2
17.4 Restrictions on OCI Calls with ALTER SESSION SET CONTAINER 17-3

17.5 Restrictions on OCI Calls with ALTER SESSION SWITCH CONTAINER SWITCH SERVICE 17-5

# 18 OCI Interface for Using Shards

18.1 About Specifying a Sharding Key and Super Sharding Key for Getting a Connection from an OCI Session Pool 18-5
18.2 About Specifying a Sharding Key and Super Sharding Key for Getting a Connection from a Custom Pool 18-7

# 19 OCI Object-Relational Programming

19.1 OCI Object Overview 19-1
19.2 About Working with Objects in OCI 19-2
    19.2.1 Basic Object Program Structure 19-3
    19.2.2 Persistent Objects, Transient Objects, and Values 19-4
        19.2.2.1 Persistent Objects 19-4
        19.2.2.2 Transient Objects 19-5
        19.2.2.3 Values 19-6
19.3 About Developing an OCI Object Application 19-6
    19.3.1 About Representing Objects in C Applications 19-8
    19.3.2 About Initializing the Environment and the Object Cache 19-9
    19.3.3 About Making Database Connections 19-10
    19.3.4 Retrieving an Object Reference from the Server 19-10
    19.3.5 Pinning an Object 19-11
        19.3.5.1 Array Pin 19-13
    19.3.6 Manipulating Object Attributes 19-13
    19.3.7 About Marking Objects and Flushing Changes 19-15
    19.3.8 Fetching Embedded Objects 19-15
    19.3.9 Object Meta-Attributes 19-17
        19.3.9.1 Persistent Object Meta-Attributes 19-17
        19.3.9.2 Additional Attribute Functions 19-19
        19.3.9.3 Transient Object Meta-Attributes 19-20
    19.3.10 Complex Object Retrieval 19-20
        19.3.10.1 About Prefetching Objects 19-22
        19.3.10.2 About Implementing Complex Object Retrieval in OCI 19-23
    19.3.11 COR Prefetching 19-24
        19.3.11.1 COR Interface 19-24
        19.3.11.2 Example of COR 19-25
    19.3.12 OCI Versus SQL Access to Objects 19-26
    19.3.13 Pin Count and Unpinning 19-28
    19.3.14 NULL Indicator Structure 19-28

19.3.15 About Creating Objects 19-31
19.3.15.1 Attribute Values of New Objects 19-31
19.3.16 About Freeing and Copying Objects 19-33
19.3.17 Object Reference and Type Reference 19-33
19.3.18 Create Objects Based on Object Views and Object Tables with Primary-Key-Based OIDs 19-34
19.3.19 Error Handling in Object Applications 19-35
19.4 About Type Inheritance 19-35
19.4.1 Substitutability 19-37
19.4.2 NOT INSTANTIABLE Types and Methods 19-37
19.4.3 OCI Support for Type Inheritance 19-38
19.4.3.1 OCIDescribeAny() 19-38
19.4.3.2 Bind and Define Functions 19-38
19.4.3.3 OCIObjectGetTypeRef() 19-39
19.4.3.4 OCIObjectCopy() 19-39
19.4.3.5 OCICollAssignElem() 19-39
19.4.3.6 OCICollAppend() 19-40
19.4.3.7 OCICollGetElem() 19-40
19.4.4 OTT Support for Type Inheritance 19-40
19.5 About Type Evolution 19-41

## 20 Object-Relational Data Types in OCI

20.1 Overview of OCI Functions for Objects 20-1
20.2 About Mapping Oracle Data Types to C 20-2
20.2.1 OCI Type Mapping Methodology 20-3
20.3 About Manipulating C Data Types with OCI 20-4
20.3.1 Precision of Oracle Number Operations 20-5
20.4 Date (OCIDate) 20-5
20.4.1 Date Example 20-6
20.5 Datetime and Interval (OCIDateTime, OCIInterval) 20-7
20.5.1 About Datetime Functions 20-8
20.5.2 Datetime Example 20-10
20.5.3 About Interval Functions 20-10
20.6 Number (OCINumber) 20-12
20.6.1 OCINumber Examples 20-12
20.7 Fixed or Variable-Length String (OCIString) 20-14
20.7.1 About String Functions 20-15
20.7.2 String Example 20-15
20.8 Raw (OCIRaw) 20-16
20.8.1 About Raw Functions 20-16
20.8.2 Raw Example 20-17

20.9 Collections (OCITable, OCIArray, OCIColl, OCIIter) 20-17
  20.9.1 Generic Collection Functions 20-18
  20.9.2 About Collection Data Manipulation Functions 20-19
  20.9.3 About Collection Scanning Functions 20-19
  20.9.4 Varray/Collection Iterator Example 20-20
  20.9.5 About Nested Table Manipulation Functions 20-21
    20.9.5.1 Nested Table Element Ordering 20-21
  20.9.6 Nested Table Locators 20-22
20.10 About Multilevel Collection Types 20-22
  20.10.1 Multilevel Collection Type Example 20-23
20.11 REF (OCIRef) 20-24
  20.11.1 About REF Manipulation Functions 20-24
  20.11.2 REF Example 20-25
20.12 Object Type Information Storage and Access 20-25
  20.12.1 Descriptor Objects 20-25
20.13 AnyType, AnyData, and AnyDataSet Interfaces 20-26
  20.13.1 About Type Interfaces 20-27
    20.13.1.1 About Creating a Parameter Descriptor for OCIType Calls 20-28
    20.13.1.2 About Obtaining the OCIType for Persistent Types 20-30
    20.13.1.3 Type Access Calls 20-30
    20.13.1.4 Extensions to OCIDescribeAny() 20-30
  20.13.2 About OCIAnyData Interfaces 20-31
  20.13.3 NCHAR Typecodes for OCIAnyData Functions 20-32
  20.13.4 About OCIAnyDataSet Interfaces 20-32
20.14 About Binding Named Data Types 20-33
  20.14.1 Named Data Type Binds 20-33
  20.14.2 About Binding REFs 20-34
  20.14.3 Information for Named Data Type and REF Binds 20-35
  20.14.4 Information Regarding Array Binds 20-35
20.15 About Defining Named Data Types 20-36
  20.15.1 About Defining Named Data Type Output Variables 20-36
  20.15.2 About Defining REF Output Variables 20-37
  20.15.3 Information for Named Data Type and REF Defines, and PL/SQL OUT Binds 20-37
    20.15.3.1 Information About Array Defines 20-39
20.16 About Binding and Defining Oracle C Data Types 20-39
  20.16.1 Bind and Define Examples 20-41
  20.16.2 Salary Update Examples 20-42
    20.16.2.1 Method 1 - Fetch, Convert, Assign 20-43
    20.16.2.2 Method 2 - Fetch and Assign 20-44
    20.16.2.3 Method 3 - Direct Fetch 20-44

20.16.2.4 Summary and Notes 20-45
20.17 SQLT_NTY Bind and Define Examples 20-45
20.17.1 SQLT_NTY Bind Example 20-46
20.17.2 SQLT_NTY Define Example 20-47

# 21 Direct Path Load Interface

21.1 Direct Path Loading Overview 21-1
21.1.1 Data Types Supported for Direct Path Loading 21-3
21.1.2 Direct Path Handles 21-4
21.1.2.1 Direct Path Context 21-4
21.1.2.2 OCI Direct Path Function Context 21-5
21.1.2.3 Direct Path Column Array and Direct Path Function Column Array 21-6
21.1.2.4 Direct Path Stream 21-6
21.1.3 About Direct Path Interface Functions 21-7
21.1.4 Limitations and Restrictions of the Direct Path Load Interface 21-8
21.1.5 Direct Path Load Examples for Scalar Columns 21-9
21.1.5.1 Data Structures Used in Direct Path Loading Example 21-9
21.1.5.2 Outline of an Example of a Direct Path Load for Scalar Columns 21-11
21.1.6 About Using a Date Cache in Direct Path Loading of Dates in OCI 21-14
21.1.6.1 OCI_ATTR_DIRPATH_DCACHE_SIZE 21-15
21.1.6.2 OCI_ATTR_DIRPATH_DCACHE_NUM 21-15
21.1.6.3 OCI_ATTR_DIRPATH_DCACHE_MISSES 21-15
21.1.6.4 OCI_ATTR_DIRPATH_DCACHE_HITS 21-15
21.1.6.5 OCI_ATTR_DIRPATH_DCACHE_DISABLE 21-15
21.1.7 About Validating Format for Oracle NUMBER and DATE Data 21-16
21.2 Direct Path Loading of Object Types 21-16
21.2.1 Direct Path Loading of Nested Tables 21-17
21.2.1.1 Describing a Nested Table Column and Its Nested Table 21-17
21.2.2 Direct Path Loading of Column Objects 21-18
21.2.2.1 Describing a Column Object 21-18
21.2.2.2 Allocating the Array Column for the Column Object 21-20
21.2.2.3 Loading Column Object Data into the Column Array 21-20
21.2.2.4 OCI_DIRPATH_COL_ERROR 21-21
21.2.3 Direct Path Loading of SQL String Columns 21-21
21.2.3.1 Describing a SQL String Column 21-22
21.2.3.2 Allocating the Column Array for SQL String Columns 21-23
21.2.3.3 Loading the SQL String Data into the Column Array 21-24
21.2.4 Direct Path Loading of REF Columns 21-24
21.2.4.1 Describing the REF Column 21-25
21.2.4.2 Allocating the Column Array for a REF Column 21-27

21.2.4.3 Loading the REF Data into the Column Array 21-28
21.2.5 Direct Path Loading of NOT FINAL Object and REF Columns 21-28
21.2.5.1 Inheritance Hierarchy 21-29
21.2.5.2 About Describing a Fixed, Derived Type to Be Loaded 21-30
21.2.5.3 About Allocating the Column Array 21-30
21.2.5.4 About Loading the Data into the Column Array 21-30
21.2.6 Direct Path Loading of Object Tables 21-30
21.2.7 Direct Path Loading a NOT FINAL Object Table 21-31
21.3 Direct Path Loading in Pieces 21-32
21.3.1 Loading Object Types in Pieces 21-33
21.4 Direct Path Context Handles and Attributes for Object Types 21-34
21.4.1 Direct Path Context Attributes 21-34
21.4.1.1 OCI_ATTR_DIRPATH_OBJ_CONSTR 21-34
21.4.2 Direct Path Function Context and Attributes 21-34
21.4.2.1 OCI_ATTR_DIRPATH_OBJ_CONSTR 21-35
21.4.2.2 OCI_ATTR_NAME 21-35
21.4.2.3 OCI_ATTR_DIRPATH_EXPR_TYPE 21-36
21.4.2.4 OCI_ATTR_DIRPATH_NO_INDEX_ERRORS 21-37
21.4.2.5 OCI_ATTR_NUM_COLS 21-37
21.4.2.6 OCI_ATTR_NUM_ROWS 21-38
21.4.3 Direct Path Column Parameter Attributes 21-39
21.4.3.1 OCI_ATTR_NAME 21-39
21.4.3.2 OCI_ATTR_DIRPATH_SID 21-41
21.4.3.3 OCI_ATTR_DIRPATH_OID 21-41
21.4.4 Direct Path Function Column Array Handle for Nonscalar Columns 21-42
21.4.4.1 OCI_ATTR_NUM_ROWS Attribute 21-42

## 22 Object Advanced Topics in OCI

22.1 Object Cache and Memory Management 22-1
22.1.1 Cache Consistency and Coherency 22-4
22.1.2 Object Cache Parameters 22-4
22.1.3 Object Cache Operations 22-5
22.1.3.1 About Pinning and Unpinning 22-5
22.1.3.2 About Freeing 22-6
22.1.3.3 About Marking and Unmarking 22-6
22.1.3.4 About Flushing 22-6
22.1.3.5 About Refreshing 22-6
22.1.4 About Loading and Removing Object Copies 22-6
22.1.4.1 About Pinning an Object Copy 22-7
22.1.4.2 About Unpinning an Object Copy 22-8

22.1.4.3 About Freeing an Object Copy 22-9
22.1.5 About Making Changes to Object Copies 22-9
22.1.5.1 About Marking an Object Copy 22-10
22.1.5.2 About Unmarking an Object Copy 22-10
22.1.6 About Synchronizing Object Copies with the Server 22-11
22.1.6.1 About Flushing Changes to the Server 22-11
22.1.6.2 About Refreshing an Object Copy 22-12
22.1.7 Object Locking 22-13
22.1.7.1 Lock Options 22-13
22.1.7.2 About Locking Objects for Update 22-13
22.1.7.3 About Locking with the NOWAIT Option 22-14
22.1.7.4 About Implementing Optimistic Locking 22-14
22.1.8 Commit and Rollback in Object Applications 22-15
22.1.9 Object Duration 22-15
22.1.9.1 Durations Example 22-16
22.1.10 Memory Layout of an Instance 22-17
22.2 Object Navigation 22-18
22.2.1 Simple Object Navigation 22-18
22.3 OCI Navigational Functions 22-21
22.3.1 About Pin/Unpin/Free Functions 22-22
22.3.2 About Flush and Refresh Functions 22-22
22.3.3 About Mark and Unmark Functions 22-22
22.3.4 About Object Meta-Attribute Accessor Functions 22-23
22.3.5 About Other Functions 22-23
22.4 Type Evolution and the Object Cache 22-23

## 23 OCI Support for XML

23.1 XML Context 23-1
23.2 XML Data on the Server 23-2
23.3 Using OCI XML DB Functions 23-2
23.4 OCI Client Access to Binary XML 23-3
23.4.1 Accessing XML Data from an OCI Application 23-4
23.4.2 Repository Context 23-4
23.4.3 Create Repository Context from a Dedicated OCI Connection 23-4
23.4.4 Create Repository Context from a Connection Pool 23-5
23.4.5 About Associating Repository Context with a Data Connection 23-5
23.4.6 About Setting XMLType Encoding Format Preference 23-6
23.4.7 Example of Using a Connection Pool 23-6

# 24 Using the Object Type Translator with OCI

24.1 What Is the Object Type Translator? 24-1
  24.1.1 About Creating Types in the Database 24-3
  24.1.2 About Invoking OTT 24-4
    24.1.2.1 Command Line 24-4
    24.1.2.2 Configuration File 24-4
    24.1.2.3 INTYPE File 24-5
24.2 OTT Command Line 24-5
  24.2.1 OTT Command-Line Invocation Example 24-5
    24.2.1.1 OTT 24-6
    24.2.1.2 USERID 24-6
    24.2.1.3 INTYPE 24-6
    24.2.1.4 OUTTYPE 24-6
    24.2.1.5 CODE 24-7
    24.2.1.6 HFILE 24-7
    24.2.1.7 INITFILE 24-7
24.3 Intype File 24-8
24.4 OTT Data Type Mappings 24-9
  24.4.1 About Mapping Object Data Types to C 24-10
  24.4.2 OTT Type Mapping Example 24-12
  24.4.3 Null Indicator Structs 24-14
  24.4.4 OTT Support for Type Inheritance 24-15
    24.4.4.1 Substitutable Object Attributes 24-17
24.5 Outtype File 24-18
24.6 About Using OTT with OCI Applications 24-19
  24.6.1 About Accessing and Manipulating Objects with OCI 24-20
  24.6.2 Calling the Initialization Function 24-21
  24.6.3 Tasks of the Initialization Function 24-23
24.7 OTT Reference 24-23
  24.7.1 OTT Command-Line Syntax 24-24
  24.7.2 OTT Parameters 24-25
    24.7.2.1 USERID 24-26
    24.7.2.2 INTYPE 24-26
    24.7.2.3 OUTTYPE 24-27
    24.7.2.4 CODE 24-27
    24.7.2.5 INITFILE 24-27
    24.7.2.6 INITFUNC 24-28
    24.7.2.7 HFILE 24-28
    24.7.2.8 CONFIG 24-28
    24.7.2.9 ERRTYPE 24-28

24.7.2.10 CASE 24-29
24.7.2.11 SCHEMA_NAMES 24-29
24.7.2.12 TRANSITIVE 24-30
24.7.2.13 URL 24-30
24.7.3 Where OTT Parameters Can Appear 24-30
24.7.4 Structure of the Intype File 24-31
24.7.4.1 Intype File Type Specifications 24-31
24.7.5 Nested Included File Generation 24-32
24.7.6 SCHEMA_NAMES Usage 24-34
24.7.6.1 Example: Schema_Names Usage 24-35
24.7.7 Default Name Mapping 24-36
24.7.8 OTT Restriction on File Name Comparison 24-37
24.7.9 OTT Command on Microsoft Windows 24-38

## 25 Oracle Database Access C API

25.1 Introduction to the Relational Functions 25-1
25.1.1 Conventions for OCI Functions 25-1
25.1.2 Purpose 25-1
25.1.3 Syntax 25-2
25.1.4 Parameters 25-2
25.1.5 Comments 25-2
25.1.6 Returns 25-2
25.1.7 Example 25-2
25.1.8 Related Functions 25-2
25.1.9 About Calling OCI Functions 25-2
25.1.10 Server Round-Trips for LOB Functions 25-3
25.2 Connect, Authorize, and Initialize Functions 25-3
25.2.1 OCIAppCtxClearAll() 25-4
25.2.2 OCIAppCtxSet() 25-5
25.2.3 OCIConnectionPoolCreate() 25-6
25.2.4 OCIConnectionPoolDestroy() 25-8
25.2.5 OCIDBShutdown() 25-9
25.2.6 OCIDBStartup() 25-10
25.2.7 OCIEnvCreate() 25-11
25.2.8 OCIEnvNlsCreate() 25-15
25.2.9 OCILogoff() 25-19
25.2.10 OCILogon() 25-20
25.2.11 OCILogon2() 25-22
25.2.12 OCIRequestDisableReplay() 25-24
25.2.13 OCIServerAttach() 25-25

25.2.14 OCIServerDetach() 25-27
25.2.15 OCISessionBegin() 25-28
25.2.16 OCISessionEnd() 25-31
25.2.17 OCISessionGet() 25-32
25.2.18 OCISessionPoolCreate() 25-39
25.2.19 OCISessionPoolDestroy() 25-43
25.2.20 OCISessionRelease() 25-43
25.2.21 OCITerminate() 25-45
25.3 Handle and Descriptor Functions 25-46
25.3.1 OCIArrayDescriptorAlloc() 25-46
25.3.2 OCIArrayDescriptorFree() 25-48
25.3.3 OCIAttrGet() 25-48
25.3.4 OCIAttrSet() 25-50
25.3.5 OCIDescriptorAlloc() 25-51
25.3.6 OCIDescriptorFree() 25-53
25.3.7 OCIHandleAlloc() 25-54
25.3.8 OCIHandleFree() 25-55
25.3.9 OCIParamGet() 25-56
25.3.10 OCIParamSet() 25-57
25.4 Bind, Define, and Describe Functions 25-59
25.4.1 OCIBindArrayOfStruct() 25-59
25.4.2 OCIBindByName() 25-61
25.4.3 OCIBindByName2() 25-66
25.4.4 OCIBindByPos() 25-72
25.4.5 OCIBindByPos2() 25-77
25.4.6 OCIBindDynamic() 25-82
25.4.7 OCIBindObject() 25-85
25.4.8 OCIDefineArrayOfStruct() 25-88
25.4.9 OCIDefineByPos() 25-89
25.4.10 OCIDefineByPos2() 25-93
25.4.11 OCIDefineDynamic() 25-98
25.4.12 OCIDefineObject() 25-101
25.4.13 OCIDescribeAny() 25-103
25.4.14 OCIStmtGetBindInfo() 25-106

## 26 More Oracle Database Access C API

26.1 Introduction to the Relational Functions 26-1
26.1.1 Conventions for OCI Functions 26-2
26.2 Statement Functions 26-2
26.2.1 OCIStmtExecute() 26-2

26.2.2 OCIStmtFetch2() 26-5
26.2.3 OCIStmtGetNextResult() 26-7
26.2.4 OCIStmtGetPieceInfo() 26-9
26.2.5 OCIStmtPrepare2() 26-10
26.2.6 OCIStmtRelease() 26-12
26.2.7 OCIStmtSetPieceInfo() 26-13
26.3 LOB Functions 26-14
26.3.1 OCIDurationBegin() 26-17
26.3.2 OCIDurationEnd() 26-18
26.3.3 OCILobAppend() 26-19
26.3.4 OCILobArrayRead() 26-20
26.3.5 OCILobArrayWrite() 26-24
26.3.6 OCILobAssign() 26-28
26.3.7 OCILobCharSetForm() 26-30
26.3.8 OCILobCharSetId() 26-31
26.3.9 OCILobClose() 26-32
26.3.10 OCILobCopy2() 26-33
26.3.11 OCILobCreateTemporary() 26-35
26.3.12 OCILobErase2() 26-37
26.3.13 OCILobFileClose() 26-38
26.3.14 OCILobFileCloseAll() 26-40
26.3.15 OCILobFileExists() 26-40
26.3.16 OCILobFileGetName() 26-42
26.3.17 OCILobFileIsOpen() 26-43
26.3.18 OCILobFileOpen() 26-44
26.3.19 OCILobFileSetName() 26-46
26.3.20 OCILobFreeTemporary() 26-47
26.3.21 OCILobGetChunkSize() 26-48
26.3.22 OCILobGetContentType() 26-49
26.3.23 OCILobGetLength2() 26-50
26.3.24 OCILobGetOptions() 26-52
26.3.25 OCILobGetStorageLimit() 26-53
26.3.26 OCILobIsEqual() 26-54
26.3.27 OCILobIsOpen() 26-55
26.3.28 OCILobIsTemporary() 26-56
26.3.29 OCILobLoadFromFile2() 26-57
26.3.30 OCILobLocatorAssign() 26-59
26.3.31 OCILobLocatorIsInit() 26-60
26.3.32 OCILobOpen() 26-62
26.3.33 OCILobRead2() 26-63
26.3.34 OCILobSetContentType() 26-67

26.3.35 OCILobSetOptions() 26-69
26.3.36 OCILobTrim2() 26-70
26.3.37 OCILobWrite2() 26-71
26.3.38 OCILobWriteAppend2() 26-75
26.4 Database Advanced Queuing and Publish-Subscribe Functions 26-79
26.4.1 OCIAQDeq() 26-79
26.4.2 OCIAQDeqArray() 26-82
26.4.3 OCIAQEnq() 26-84
26.4.4 OCIAQEnqArray() 26-86
26.4.5 OCIAQListen2() 26-88
26.4.6 OCISubscriptionDisable() 26-89
26.4.7 OCISubscriptionEnable() 26-90
26.4.8 OCISubscriptionPost() 26-92
26.4.9 OCISubscriptionRegister() 26-93
26.4.10 OCISubscriptionUnRegister() 26-95
26.5 Direct Path Loading Functions 26-97
26.5.1 OCIDirPathAbort() 26-97
26.5.2 OCIDirPathColArrayEntryGet() 26-98
26.5.3 OCIDirPathColArrayEntrySet() 26-99
26.5.4 OCIDirPathColArrayReset() 26-101
26.5.5 OCIDirPathColArrayRowGet() 26-101
26.5.6 OCIDirPathColArrayToStream() 26-102
26.5.7 OCIDirPathDataSave() 26-104
26.5.8 OCIDirPathFinish() 26-105
26.5.9 OCIDirPathFlushRow() 26-106
26.5.10 OCIDirPathLoadStream() 26-107
26.5.11 OCIDirPathPrepare() 26-108
26.5.12 OCIDirPathStreamReset() 26-109
26.6 Thread Management Functions 26-110
26.6.1 OCIThreadClose() 26-111
26.6.2 OCIThreadCreate() 26-112
26.6.3 OCIThreadHandleGet() 26-113
26.6.4 OCIThreadHndDestroy() 26-114
26.6.5 OCIThreadHndInit() 26-114
26.6.6 OCIThreadIdDestroy() 26-115
26.6.7 OCIThreadIdGet() 26-116
26.6.8 OCIThreadIdInit() 26-117
26.6.9 OCIThreadIdNull() 26-118
26.6.10 OCIThreadIdSame() 26-119
26.6.11 OCIThreadIdSet() 26-120
26.6.12 OCIThreadIdSetNull() 26-121

26.6.13 OCIThreadInit() 26-122
26.6.14 OCIThreadIsMulti() 26-122
26.6.15 OCIThreadJoin() 26-123
26.6.16 OCIThreadKeyDestroy() 26-124
26.6.17 OCIThreadKeyGet() 26-125
26.6.18 OCIThreadKeyInit() 26-126
26.6.19 OCIThreadKeySet() 26-127
26.6.20 OCIThreadMutexAcquire() 26-128
26.6.21 OCIThreadMutexDestroy() 26-128
26.6.22 OCIThreadMutexInit() 26-129
26.6.23 OCIThreadMutexRelease() 26-130
26.6.24 OCIThreadProcessInit() 26-131
26.6.25 OCIThreadTerm() 26-132
26.7 Transaction Functions 26-132
26.7.1 OCITransCommit() 26-133
26.7.2 OCITransDetach() 26-135
26.7.3 OCITransForget() 26-136
26.7.4 OCITransMultiPrepare() 26-137
26.7.5 OCITransPrepare() 26-138
26.7.6 OCITransRollback() 26-139
26.7.7 OCITransStart() 26-140
26.8 Sharding Functions 26-145
26.8.1 OCIShardingKeyColumnAdd() 26-146
26.8.2 OCIShardingKeyReset() 26-148
26.8.3 OCIShardInstancesGet() 26-149
26.9 Miscellaneous Functions 26-152
26.9.1 OCIBreak() 26-152
26.9.2 OCIClientVersion() 26-153
26.9.3 OCIErrorGet() 26-155
26.9.4 OCILdaToSvcCtx() 26-157
26.9.5 OCIPasswordChange() 26-158
26.9.6 OCIPing() 26-160
26.9.7 OCIReset() 26-161
26.9.8 OCIRowidToChar() 26-162
26.9.9 OCIServerRelease() 26-163
26.9.10 OCIServerRelease2() 26-164
26.9.11 OCIServerVersion() 26-166
26.9.12 OCISvcCtxToLda() 26-167
26.9.13 OCIUserCallbackGet() 26-168
26.9.14 OCIUserCallbackRegister() 26-169

## 27 OCI Navigational and Type Functions

27.1 Introduction to the Navigational and Type Functions 27-1
  27.1.1 Object Types and Lifetimes 27-2
  27.1.2 Terminology 27-3
  27.1.3 Conventions for OCI Functions 27-3
  27.1.4 Return Values 27-3
  27.1.5 Navigational Function Return Values 27-4
  27.1.6 Server Round-Trips for Cache and Object Functions 27-4
  27.1.7 Navigational Function Error Codes 27-4
27.2 OCI Flush or Refresh Functions 27-6
  27.2.1 OCICacheFlush() 27-6
  27.2.2 OCICacheRefresh() 27-8
  27.2.3 OCIObjectFlush() 27-10
  27.2.4 OCIObjectRefresh() 27-11
27.3 OCI Mark or Unmark Object and Cache Functions 27-13
  27.3.1 OCICacheUnmark() 27-13
  27.3.2 OCIObjectMarkDelete() 27-14
  27.3.3 OCIObjectMarkDeleteByRef() 27-15
  27.3.4 OCIObjectMarkUpdate() 27-16
  27.3.5 OCIObjectUnmark() 27-18
  27.3.6 OCIObjectUnmarkByRef() 27-19
27.4 OCI Get Object Status Functions 27-20
  27.4.1 OCIObjectExists() 27-20
  27.4.2 OCIObjectGetProperty() 27-21
  27.4.3 OCIObjectIsDirty() 27-24
  27.4.4 OCIObjectIsLocked() 27-25
27.5 OCI Miscellaneous Object Functions 27-26
  27.5.1 OCIObjectCopy() 27-27
  27.5.2 OCIObjectGetAttr() 27-29
  27.5.3 OCIObjectGetInd() 27-31
  27.5.4 OCIObjectGetObjectRef() 27-32
  27.5.5 OCIObjectGetTypeRef() 27-33
  27.5.6 OCIObjectLock() 27-34
  27.5.7 OCIObjectLockNoWait() 27-35
  27.5.8 OCIObjectNew() 27-36
  27.5.9 OCIObjectSetAttr() 27-40
27.6 OCI Pin, Unpin, and Free Functions 27-42
  27.6.1 OCICacheFree() 27-42
  27.6.2 OCICacheUnpin() 27-44
  27.6.3 OCIObjectArrayPin() 27-45

27.6.4 OCIObjectFree() 27-46
27.6.5 OCIObjectPin() 27-48
27.6.6 OCIObjectPinCountReset() 27-50
27.6.7 OCIObjectPinTable() 27-51
27.6.8 OCIObjectUnpin() 27-53
27.7 OCI Type Information Accessor Functions 27-55
27.7.1 OCITypeArrayByName() 27-55
27.7.2 OCITypeArrayByFullName() 27-58
27.7.3 OCITypeArrayByRef() 27-60
27.7.4 OCITypeByFullName() 27-62
27.7.5 OCITypeByName() 27-65
27.7.6 OCITypeByRef() 27-67
27.7.7 OCITypePackage() 27-69

## 28 OCI Data Type Mapping and Manipulation Functions

28.1 Introduction to Data Type Mapping and Manipulation Functions 28-1
28.1.1 Conventions for OCI Functions 28-2
28.1.2 Returns 28-2
28.1.3 Data Type Mapping and Manipulation Function Return Values 28-2
28.1.4 Functions Returning Other Values 28-2
28.1.5 Server Round-Trips for Data Type Mapping and Manipulation Functions 28-3
28.1.6 Examples 28-3
28.2 OCI Collection and Iterator Functions 28-3
28.2.1 OCICollAppend() 28-4
28.2.2 OCICollAssign() 28-5
28.2.3 OCICollAssignElem() 28-6
28.2.4 OCICollGetElem() 28-7
28.2.5 OCICollGetElemArray() 28-10
28.2.6 OCICollIsLocator() 28-12
28.2.7 OCICollMax() 28-13
28.2.8 OCICollSize() 28-14
28.2.9 OCICollTrim() 28-15
28.2.10 OCIIterCreate() 28-16
28.2.11 OCIIterDelete() 28-18
28.2.12 OCIIterGetCurrent() 28-19
28.2.13 OCIIterInit() 28-20
28.2.14 OCIIterNext() 28-22
28.2.15 OCIIterPrev() 28-23
28.3 OCI Date, Datetime, and Interval Functions 28-25
28.3.1 OCIDateAddDays() 28-27

28.3.2 OCIDateAddMonths() 28-28
28.3.3 OCIDateAssign() 28-29
28.3.4 OCIDateCheck() 28-29
28.3.5 OCIDateCompare() 28-31
28.3.6 OCIDateDaysBetween() 28-31
28.3.7 OCIDateFromText() 28-32
28.3.8 OCIDateGetDate() 28-33
28.3.9 OCIDateGetTime() 28-34
28.3.10 OCIDateLastDay() 28-35
28.3.11 OCIDateNextDay() 28-36
28.3.12 OCIDateSetDate() 28-37
28.3.13 OCIDateSetTime() 28-37
28.3.14 OCIDateSysDate() 28-38
28.3.15 OCIDateTimeAssign() 28-39
28.3.16 OCIDateTimeCheck() 28-40
28.3.17 OCIDateTimeCompare() 28-41
28.3.18 OCIDateTimeConstruct() 28-42
28.3.19 OCIDateTimeConvert() 28-44
28.3.20 OCIDateTimeFromArray() 28-44
28.3.21 OCIDateTimeFromText() 28-46
28.3.22 OCIDateTimeGetDate() 28-47
28.3.23 OCIDateTimeGetTime() 28-48
28.3.24 OCIDateTimeGetTimeZoneName() 28-49
28.3.25 OCIDateTimeGetTimeZoneOffset() 28-50
28.3.26 OCIDateTimeIntervalAdd() 28-51
28.3.27 OCIDateTimeIntervalSub() 28-52
28.3.28 OCIDateTimeSubtract() 28-53
28.3.29 OCIDateTimeSysTimeStamp() 28-54
28.3.30 OCIDateTimeToArray() 28-54
28.3.31 OCIDateTimeToText() 28-55
28.3.32 OCIDateToText() 28-57
28.3.33 OCIDateZoneToZone() 28-58
28.3.34 OCIIntervalAdd() 28-59
28.3.35 OCIIntervalAssign() 28-60
28.3.36 OCIIntervalCheck() 28-61
28.3.37 OCIIntervalCompare() 28-62
28.3.38 OCIIntervalDivide() 28-63
28.3.39 OCIIntervalFromNumber() 28-64
28.3.40 OCIIntervalFromText() 28-65
28.3.41 OCIIntervalFromTZ() 28-66
28.3.42 OCIIntervalGetDaySecond() 28-67

28.3.43 OCIIntervalGetYearMonth() 28-68
28.3.44 OCIIntervalMultiply() 28-69
28.3.45 OCIIntervalSetDaySecond() 28-69
28.3.46 OCIIntervalSetYearMonth() 28-70
28.3.47 OCIIntervalSubtract() 28-71
28.3.48 OCIIntervalToNumber() 28-72
28.3.49 OCIIntervalToText() 28-73
28.4 OCI NUMBER Functions 28-74
28.4.1 OCINumberAbs() 28-76
28.4.2 OCINumberAdd() 28-77
28.4.3 OCINumberArcCos() 28-77
28.4.4 OCINumberArcSin() 28-78
28.4.5 OCINumberArcTan() 28-79
28.4.6 OCINumberArcTan2() 28-79
28.4.7 OCINumberAssign() 28-80
28.4.8 OCINumberCeil() 28-81
28.4.9 OCINumberCmp() 28-82
28.4.10 OCINumberCos() 28-82
28.4.11 OCINumberDec() 28-83
28.4.12 OCINumberDiv() 28-84
28.4.13 OCINumberExp() 28-85
28.4.14 OCINumberFloor() 28-85
28.4.15 OCINumberFromInt() 28-86
28.4.16 OCINumberFromReal() 28-87
28.4.17 OCINumberFromText() 28-88
28.4.18 OCINumberHypCos() 28-89
28.4.19 OCINumberHypSin() 28-90
28.4.20 OCINumberHypTan() 28-90
28.4.21 OCINumberInc() 28-91
28.4.22 OCINumberIntPower() 28-92
28.4.23 OCINumberIsInt() 28-93
28.4.24 OCINumberIsZero() 28-94
28.4.25 OCINumberLn() 28-94
28.4.26 OCINumberLog() 28-95
28.4.27 OCINumberMod() 28-96
28.4.28 OCINumberMul() 28-97
28.4.29 OCINumberNeg() 28-97
28.4.30 OCINumberPower() 28-98
28.4.31 OCINumberPrec() 28-99
28.4.32 OCINumberRound() 28-100
28.4.33 OCINumberSetPi() 28-100

28.4.34 OCINumberSetZero() 28-101
28.4.35 OCINumberShift() 28-102
28.4.36 OCINumberSign() 28-102
28.4.37 OCINumberSin() 28-103
28.4.38 OCINumberSqrt() 28-104
28.4.39 OCINumberSub() 28-105
28.4.40 OCINumberTan() 28-105
28.4.41 OCINumberToInt() 28-106
28.4.42 OCINumberToReal() 28-107
28.4.43 OCINumberToRealArray() 28-108
28.4.44 OCINumberToText() 28-109
28.4.45 OCINumberTrunc() 28-110
28.5 OCI Raw Functions 28-111
28.5.1 OCIRawAllocSize() 28-111
28.5.2 OCIRawAssignBytes() 28-112
28.5.3 OCIRawAssignRaw() 28-113
28.5.4 OCIRawPtr() 28-114
28.5.5 OCIRawResize() 28-115
28.5.6 OCIRawSize() 28-116
28.6 OCI REF Functions 28-117
28.6.1 OCIRefAssign() 28-117
28.6.2 OCIRefClear() 28-118
28.6.3 OCIRefFromHex() 28-119
28.6.4 OCIRefHexSize() 28-120
28.6.5 OCIRefIsEqual() 28-121
28.6.6 OCIRefIsNull() 28-122
28.6.7 OCIRefToHex() 28-123
28.7 OCI String Functions 28-125
28.7.1 OCIStringAllocSize() 28-125
28.7.2 OCIStringAssign() 28-126
28.7.3 OCIStringAssignText() 28-127
28.7.4 OCIStringPtr() 28-128
28.7.5 OCIStringResize() 28-129
28.7.6 OCIStringSize() 28-130
28.8 OCI Table Functions 28-131
28.8.1 OCITableDelete() 28-131
28.8.2 OCITableExists() 28-132
28.8.3 OCITableFirst() 28-133
28.8.4 OCITableLast() 28-135
28.8.5 OCITableNext() 28-136
28.8.6 OCITablePrev() 28-137

28.8.7 OCITableSize() 28-138

# 29 OCI Cartridge Functions

29.1 Introduction to External Procedure and Cartridge Services Functions 29-1
29.1.1 Conventions for OCI Functions 29-1
29.2 Cartridge Services — OCI External Procedures 29-2
29.2.1 OCIExtProcAllocCallMemory() 29-2
29.2.2 OCIExtProcGetEnv() 29-3
29.2.3 OCIExtProcRaiseExcp() 29-4
29.2.4 OCIExtProcRaiseExcpWithMsg() 29-5
29.3 Cartridge Services — Memory Services 29-6
29.3.1 OCIDurationBegin() 29-7
29.3.2 OCIDurationEnd() 29-8
29.3.3 OCIMemoryAlloc() 29-8
29.3.4 OCIMemoryFree() 29-9
29.3.5 OCIMemoryResize() 29-10
29.4 Cartridge Services — Maintaining Context 29-11
29.4.1 OCIContextClearValue() 29-11
29.4.2 OCIContextGenerateKey() 29-12
29.4.3 OCIContextGetValue() 29-13
29.4.4 OCIContextSetValue() 29-13
29.5 Cartridge Services — Parameter Manager Interface 29-15
29.5.1 OCIExtractFromFile() 29-15
29.5.2 OCIExtractFromList() 29-16
29.5.3 OCIExtractFromStr() 29-17
29.5.4 OCIExtractInit() 29-18
29.5.5 OCIExtractReset() 29-19
29.5.6 OCIExtractSetKey() 29-20
29.5.7 OCIExtractSetNumKeys() 29-21
29.5.8 OCIExtractTerm() 29-22
29.5.9 OCIExtractToBool() 29-22
29.5.10 OCIExtractToInt() 29-23
29.5.11 OCIExtractToList() 29-24
29.5.12 OCIExtractToOCINum() 29-25
29.5.13 OCIExtractToStr() 29-25
29.6 Cartridge Services — File I/O Interface 29-26
29.6.1 OCIFileClose() 29-27
29.6.2 OCIFileExists() 29-28
29.6.3 OCIFileFlush() 29-29
29.6.4 OCIFileGetLength() 29-29

29.6.5 OCIFileInit() 29-30
29.6.6 OCIFileOpen() 29-31
29.6.7 OCIFileRead() 29-32
29.6.8 OCIFileSeek() 29-33
29.6.9 OCIFileTerm() 29-34
29.6.10 OCIFileWrite() 29-34
29.7 Cartridge Services — String Formatting Interface 29-35
29.7.1 OCIFormatInit() 29-36
29.7.2 OCIFormatString() 29-36
29.7.3 OCIFormatTerm() 29-41

## 30 OCI Any Type and Data Functions

30.1 Introduction to Any Type and Data Interfaces 30-1
30.1.1 Conventions for OCI Functions 30-1
30.2 OCI Type Interface Functions 30-2
30.2.1 OCITypeAddAttr() 30-2
30.2.2 OCITypeBeginCreate() 30-3
30.2.3 OCITypeEndCreate() 30-5
30.2.4 OCITypeSetBuiltin() 30-5
30.2.5 OCITypeSetCollection() 30-6
30.3 OCI Any Data Interface Functions 30-7
30.3.1 OCIAnyDataAccess() 30-8
30.3.2 OCIAnyDataAttrGet() 30-9
30.3.3 OCIAnyDataAttrSet() 30-11
30.3.4 OCIAnyDataBeginCreate() 30-13
30.3.5 OCIAnyDataCollAddElem() 30-15
30.3.6 OCIAnyDataCollGetElem() 30-17
30.3.7 OCIAnyDataConvert() 30-18
30.3.8 OCIAnyDataDestroy() 30-20
30.3.9 OCIAnyDataEndCreate() 30-20
30.3.10 OCIAnyDataGetCurrAttrNum() 30-21
30.3.11 OCIAnyDataGetType() 30-22
30.3.12 OCIAnyDataIsNull() 30-22
30.3.13 OCIAnyDataTypeCodeToSqlt() 30-23
30.4 OCI Any Data Set Interface Functions 30-24
30.4.1 OCIAnyDataSetAddInstance() 30-25
30.4.2 OCIAnyDataSetBeginCreate() 30-26
30.4.3 OCIAnyDataSetDestroy() 30-27
30.4.4 OCIAnyDataSetEndCreate() 30-27
30.4.5 OCIAnyDataSetGetCount() 30-28

30.4.6 OCIAnyDataSetGetInstance() 30-29
30.4.7 OCIAnyDataSetGetType() 30-29

# 31 OCI Globalization Support Functions

31.1 Introduction to Globalization Support in OCI 31-1
31.1.1 Conventions for OCI Functions 31-1
31.2 OCI Locale Functions 31-2
31.2.1 OCINlsCharSetIdToName() 31-2
31.2.2 OCINlsCharSetNameToId() 31-3
31.2.3 OCINlsEnvironmentVariableGet() 31-4
31.2.4 OCINlsGetInfo() 31-5
31.2.5 OCINlsNumericInfoGet() 31-7
31.3 OCI Locale-Mapping Function 31-8
31.3.1 OCINlsNameMap() 31-9
31.4 OCI String Manipulation Functions 31-10
31.4.1 OCIMultiByteInSizeToWideChar() 31-12
31.4.2 OCIMultiByteStrCaseConversion() 31-13
31.4.3 OCIMultiByteStrcat() 31-14
31.4.4 OCIMultiByteStrcmp() 31-15
31.4.5 OCIMultiByteStrcpy() 31-16
31.4.6 OCIMultiByteStrlen() 31-16
31.4.7 OCIMultiByteStrncat() 31-17
31.4.8 OCIMultiByteStrncmp() 31-18
31.4.9 OCIMultiByteStrncpy() 31-19
31.4.10 OCIMultiByteStrnDisplayLength() 31-20
31.4.11 OCIMultiByteToWideChar() 31-21
31.4.12 OCIWideCharInSizeToMultiByte() 31-22
31.4.13 OCIWideCharMultiByteLength() 31-23
31.4.14 OCIWideCharStrCaseConversion() 31-23
31.4.15 OCIWideCharStrcat() 31-24
31.4.16 OCIWideCharStrchr() 31-25
31.4.17 OCIWideCharStrcmp() 31-26
31.4.18 OCIWideCharStrcpy() 31-27
31.4.19 OCIWideCharStrlen() 31-28
31.4.20 OCIWideCharStrncat() 31-28
31.4.21 OCIWideCharStrncmp() 31-29
31.4.22 OCIWideCharStrncpy() 31-30
31.4.23 OCIWideCharStrrchr() 31-31
31.4.24 OCIWideCharToLower() 31-32
31.4.25 OCIWideCharToMultiByte() 31-33

31.4.26 OCIWideCharToUpper() 31-34
31.5 OCI Character Classification Functions 31-34
31.5.1 OCIWideCharIsAlnum() 31-35
31.5.2 OCIWideCharIsAlpha() 31-36
31.5.3 OCIWideCharIsCntrl() 31-36
31.5.4 OCIWideCharIsDigit() 31-37
31.5.5 OCIWideCharIsGraph() 31-37
31.5.6 OCIWideCharIsLower() 31-38
31.5.7 OCIWideCharIsPrint() 31-38
31.5.8 OCIWideCharIsPunct() 31-39
31.5.9 OCIWideCharIsSingleByte() 31-39
31.5.10 OCIWideCharIsSpace() 31-40
31.5.11 OCIWideCharIsUpper() 31-40
31.5.12 OCIWideCharIsXdigit() 31-41
31.6 OCI Character Set Conversion Functions 31-41
31.6.1 OCICharSetConversionIsReplacementUsed() 31-42
31.6.2 OCICharSetToUnicode() 31-42
31.6.3 OCINlsCharSetConvert() 31-43
31.6.4 OCIUnicodeToCharSet() 31-45
31.7 OCI Messaging Functions 31-46
31.7.1 OCIMessageClose() 31-46
31.7.2 OCIMessageGet() 31-47
31.7.3 OCIMessageOpen() 31-48

## 32 OCI XML DB Functions

32.1 Introduction to XML DB Support in OCI 32-1
32.1.1 Conventions for OCI Functions 32-1
32.1.2 Returns 32-1
32.2 OCI XML DB Functions 32-2
32.2.1 OCIBinXmlCreateReposCtxFromConn() 32-2
32.2.2 OCIBinXmlCreateReposCtxFromCPool() 32-3
32.2.3 OCIBinXmlSetFormatPref() 32-4
32.2.4 OCIBinXmlSetReposCtxForConn() 32-4
32.2.5 OCIXmlDbFreeXmlCtx() 32-5
32.2.6 OCIXmlDbInitXmlCtx() 32-5

## 33 Oracle ODBC Driver

## 34 Introduction to the OCI Interface for XStream

34.1 About the XStream Interface 34-1

  34.1.1 XStream Out 34-1

  34.1.2 XStream In 34-2

  34.1.3 Position Order and LCR Streams 34-2

  34.1.4 XStream and Character Sets 34-2

34.2 Handler and Descriptor Attributes 34-3

  34.2.1 Conventions 34-3

  34.2.2 Server Handle Attributes 34-3

    34.2.2.1 OCI_ATTR_XSTREAM_ACK_INTERVAL 34-3

    34.2.2.2 OCI_ATTR_XSTREAM_IDLE_TIMEOUT 34-4

## 35 OCI XStream Functions

35.1 About Using the XStream Interface 35-1

  35.1.1 XStream Out 35-2

    35.1.1.1 LCR Streams 35-2

    35.1.1.2 The Processed Low Position and Restart Considerations 35-3

  35.1.2 XStream In 35-3

    35.1.2.1 Processed Low Position and Restart Ability 35-4

    35.1.2.2 Stream Position 35-4

  35.1.3 Security of XStreams 35-4

35.2 Introduction to XStream Functions 35-5

35.3 OCI XStream Functions 35-5

  35.3.1 OCILCRAttributesGet() 35-7

  35.3.2 OCILCRAttributesSet() 35-9

  35.3.3 OCILCRComparePosition() 35-10

  35.3.4 OCILCRConvertPosition() 35-11

  35.3.5 OCILCRFree() 35-12

  35.3.6 OCILCRDDLInfoGet() 35-13

  35.3.7 OCILCRHeaderGet() 35-15

  35.3.8 OCILCRRowStmtGet() 35-17

  35.3.9 OCILCRRowStmtWithBindVarGet() 35-18

  35.3.10 OCILCRNew() 35-20

  35.3.11 OCILCRRowColumnInfoGet() 35-21

  35.3.12 OCILCRRowColumnInfoSet() 35-24

  35.3.13 OCILCRDDLInfoSet() 35-26

  35.3.14 OCILCRGetLCRIDVersion() 35-29

  35.3.15 OCILCRHeaderSet() 35-29

  35.3.16 OCILCRLobInfoGet() 35-32

  35.3.17 OCILCRLobInfoSet() 35-33

35.3.18 OCILCRSCNsFromPosition() 35-34
35.3.19 OCILCRSCNToPosition() 35-35
35.3.20 OCILCRScnToPosition2() 35-36
35.3.21 OCILCRWhereClauseGet() 35-37
35.3.22 OCILCRWhereClauseWithBindVarGet() 35-38
35.3.23 OCIXStreamInAttach() 35-40
35.3.24 OCIXStreamInDetach() 35-41
35.3.25 OCIXStreamInLCRSend() 35-42
35.3.26 OCIXStreamInLCRCallbackSend() 35-43
35.3.27 OCIXStreamInProcessedLWMGet() 35-48
35.3.28 OCIXStreamInErrorGet() 35-49
35.3.29 OCIXStreamInFlush() 35-50
35.3.30 OCIXStreamInChunkSend() 35-51
35.3.31 OCIXStreamInCommit() 35-54
35.3.32 OCIXStreamInSessionSet() 35-55
35.3.33 OCIXStreamOutAttach() 35-56
35.3.34 OCIXStreamOutDetach() 35-58
35.3.35 OCIXStreamOutLCRReceive() 35-58
35.3.36 OCIXStreamOutLCRCallbackReceive() 35-60
35.3.37 OCIXStreamOutProcessedLWMSet() 35-65
35.3.38 OCIXStreamOutChunkReceive() 35-66
35.3.39 OCIXStreamOutGetNextChunk() 35-69
35.3.40 OCIXStreamOutSessionSet() 35-72

## 36 OCI SODA Functions

36.1 Introduction to OCI SODA Functions 36-1
36.2 OCI SODA Functions 36-1
36.2.1 OCISodaBulkInsert() 36-3
36.2.2 OCISodaBulkInsertAndGet() 36-4
36.2.3 OCISodaBulkInsertAndGetWithCtnt() 36-6
36.2.4 OCISodaBulkInsertWithCtnt() 36-9
36.2.5 OCISodaCollCreate() 36-11
36.2.6 OCISodaCollCreateWithMetadata() 36-13
36.2.7 OCISodaCollDrop() 36-14
36.2.8 OCISodaCollGetNext() 36-16
36.2.9 OCISodaCollList() 36-17
36.2.10 OCISodaCollOpen() 36-18
36.2.11 OCISodaDataGuideGet() 36-20
36.2.12 OCISodaDocCount() 36-22
36.2.13 OCISodaDocCountWithFilter() 36-24

36.2.14 OCISodaDocCreate() 36-25
36.2.15 OCISodaDocCreateWithKey() 36-27
36.2.16 OCISodaDocCreateWithKeyAndMType() 36-28
36.2.17 OCISodaDocGetNext() 36-30
36.2.18 OCISodaFind() 36-31
36.2.19 OCISodaFindOne() 36-34
36.2.20 OCISodaFindOneWithKey() 36-35
36.2.21 OCISodaIndexCreate() 36-37
36.2.22 OCISodaIndexDrop() 36-39
36.2.23 OCISodaInsert() 36-40
36.2.24 OCISodaInsertAndGet() 36-41
36.2.25 OCISodaInsertAndGetWithCtnt() 36-43
36.2.26 OCISodaInsertWithCtnt() 36-45
36.2.27 OCISodaRemove() 36-48
36.2.28 OCISodaRemoveOneWithKey() 36-49
36.2.29 OCISodaReplOne() 36-50
36.2.30 OCISodaReplOneAndGet() 36-52
36.2.31 OCISodaReplOneAndGetWithCtnt() 36-54
36.2.32 OCISodaReplOneAndGetWithKey() 36-57
36.2.33 OCISodaReplOneWithCtnt() 36-59
36.2.34 OCISodaReplOneWithKey() 36-62
36.2.35 OCISodaOperKeysSet() 36-63

## A Handle and Descriptor Attributes

A.1 Conventions A-2
A.2 Environment Handle Attributes A-3
A.3 Error Handle Attributes A-10
A.4 Service Context Handle Attributes A-10
A.5 Server Handle Attributes A-15
A.5.1 Authentication Information Handle Attributes A-20
A.5.2 User Session Handle Attributes A-21
A.6 Administration Handle Attributes A-34
A.7 Connection Pool Handle Attributes A-34
A.7.1 Session Pool Handle Attributes A-36
A.8 Transaction Handle Attributes A-40
A.9 Statement Handle Attributes A-41
A.10 Bind Handle Attributes A-51
A.11 Define Handle Attributes A-54
A.12 Describe Handle Attributes A-57
A.13 Parameter Descriptor Attributes A-58

A.14 Shard Instance Descriptor Attributes A-59
A.15 SODA Document Handle Attributes A-60
A.16 SODA Collection Handle Attributes A-62
A.17 SODA Output Options Handle Attributes A-71
A.18 SODA Operation Options Handle Attributes A-71
A.19 LOB Descriptor and LOB Locator Attributes A-73
A.20 Complex Object Attributes A-74
  A.20.1 Complex Object Retrieval Handle Attributes A-74
  A.20.2 Complex Object Retrieval Descriptor Attributes A-74
A.21 Database Advanced Queuing Descriptor Attributes A-75
  A.21.1 OCIAQEnqOptions Descriptor Attributes A-75
  A.21.2 OCIAQDeqOptions Descriptor Attributes A-77
  A.21.3 OCIAQMsgProperties Descriptor Attributes A-81
  A.21.4 OCIAQAgent Descriptor Attributes A-85
  A.21.5 OCIServerDNs Descriptor Attributes A-85
A.22 Subscription Handle Attributes A-86
  A.22.1 Continuous Query Notification Attributes A-92
  A.22.2 Continuous Query Notification Descriptor Attributes A-93
  A.22.3 Notification Descriptor Attributes A-96
  A.22.4 Invalidated Query Attributes A-97
A.23 Direct Path Loading Handle Attributes A-98
  A.23.1 Direct Path Context Handle (OCIDirPathCtx) Attributes A-99
  A.23.2 Direct Path Function Context Handle (OCIDirPathFuncCtx) Attributes A-106
  A.23.3 Direct Path Function Column Array Handle (OCIDirPathColArray) Attributes A-108
  A.23.4 Direct Path Stream Handle (OCIDirPathStream) Attributes A-109
  A.23.5 Direct Path Column Parameter Attributes A-110
    A.23.5.1 About Accessing Column Parameter Attributes A-110
A.24 Process Handle Attributes A-114
A.25 Event Handle Attributes A-116

## B OCI Demonstration Programs

## C OCI Function Server Round-Trips

C.1 Relational Function Round-Trips C-2
C.2 LOB Function Round-Trips C-3
C.3 Object and Cache Function Round-Trips C-5
C.4 Describe Operation Round-Trips C-6
C.5 Data Type Mapping and Manipulation Function Round-Trips C-7
C.6 Any Type and Data Function Round-Trips C-7

C.7 Other Local Functions C-7

## D Getting Started with OCI for Windows

D.1 What Is Included in the OCI Package for Windows? D-1
D.2 Oracle Directory Structure for Windows D-2
D.3 Sample OCI Programs for Windows D-2
D.4 About Compiling OCI Applications for Windows D-2
D.5 About Linking OCI Applications for Windows D-3
  D.5.1 oci.lib D-3
  D.5.2 Client DLL Loading When Using Load Library() D-3
D.6 About Running OCI Applications for Windows D-4
D.7 Oracle XA Library D-4
  D.7.1 About Compiling and Linking an OCI Program with the Oracle XA Library D-5
  D.7.2 About Using XA Dynamic Registration D-5
    D.7.2.1 Adding an Environmental Variable for the Current Session D-5
    D.7.2.2 About Adding a Registry Variable for All Sessions D-5
    D.7.2.3 Adding a Registry Variable: D-6
  D.7.3 XA and TP Monitor Information D-6
D.8 About Using the Object Type Translator for Windows D-6

## E Deprecated OCI Features and Functions

E.1 Deprecated Initialize Functions E-1
  E.1.1 OCIEnvInit() E-2
  E.1.2 OCIInitialize() E-3
E.2 Deprecated Statement Functions E-6
  E.2.1 OCIStmtFetch() E-6
  E.2.2 OCIStmtPrepare() E-7
E.3 Deprecated Lob Functions E-9
  E.3.1 OCILobCopy() E-9
  E.3.2 OCILobDisableBuffering() E-10
  E.3.3 OCILobEnableBuffering() E-11
  E.3.4 OCILobErase() E-12
  E.3.5 OCILobFlushBuffer() E-13
  E.3.6 OCILobGetLength() E-14
  E.3.7 OCILobLoadFromFile() E-15
  E.3.8 OCILobRead() E-15
  E.3.9 OCILobTrim() E-19
  E.3.10 OCILobWrite() E-20
  E.3.11 OCILobWriteAppend() E-24

E.4 Deprecated Database Advanced Queuing Functions E-27
  E.4.1 OCIAQListen() E-27

# F Multithreaded extproc Agent

F.1 Why Use the Multithreaded extproc Agent? F-1
  F.1.1 The Challenge of Dedicated Agent Architecture F-1
  F.1.2 The Advantage of Multithreading F-1
F.2 Multithreaded extproc Agent Architecture F-2
  F.2.1 Monitor Thread F-4
  F.2.2 Dispatcher Threads F-4
  F.2.3 Task Threads F-4
F.3 Administering the Multithreaded extproc Agent F-5
  F.3.1 Agent Control Utility (agtctl) Commands F-5
  F.3.2 About Using agtctl in Single-Line Command Mode F-6
    F.3.2.1 Setting Configuration Parameters for a Multithreaded extproc Agent F-7
    F.3.2.2 Starting a Multithreaded extproc Agent F-7
    F.3.2.3 Shutting Down a Multithreaded extproc Agent F-7
    F.3.2.4 Examining the Value of Configuration Parameters F-8
    F.3.2.5 Resetting a Configuration Parameter to Its Default Value F-8
    F.3.2.6 Deleting an Entry for a Specific SID from the Control File F-8
    F.3.2.7 Requesting Help F-9
  F.3.3 Using Shell Mode Commands F-9
    F.3.3.1 Example: Setting a Configuration Parameter F-9
    F.3.3.2 Example: Starting a Multithreaded extproc Agent F-10
  F.3.4 Configuration Parameters for Multithreaded extproc Agent Control F-10

# Index

# List of Examples

4-1 Using the OCI_ATTR_USERNAME Attribute to Set the User Name in the Session Handle 4-11
4-2 Returning Describe Information in the Statement Handle Relating to Select-List Items 4-12
4-3 Using the OCILogon2 Call for a Single User Session 4-20
4-4 Enabling a Local User to Serve as a Proxy for Another User 4-22
4-5 Connection String to Use for the Proxy User 4-22
4-6 Preserving Case Sensitivity When Enabling a Local User to Serve as a Proxy for Another User 4-22
4-7 Preserving Case Sensitivity in the Connection String 4-23
4-8 Using "dilbert[mybert]" in the Connection String 4-23
4-9 Using "dilbert[mybert]"["joe[myjoe]"] in the Connection String 4-23
4-10 Setting the Target User Name 4-23
4-11 Using OCI to Set the OCI_ATTR_PROXY_CLIENT Attribute and the Proxy dilbert 4-23
4-12 Creating and Initializing an OCI Environment 4-24
4-13 Getting Locale Information in OCI 4-43
4-14 Basic String Manipulation in OCI 4-44
4-15 Classifying Characters in OCI 4-45
4-16 Converting Character Sets in OCI 4-46
4-17 Retrieving a Message from a Text Message File 4-47
5-1 OCI Bind and Define Support for 64-Bit Integers 5-13
5-2 Binding 8-Byte Integer Data Types for OUT Binds of a DML Returning Statement 5-14
6-1 Binding Both Input and Output Variables in Nonquery Operations 6-6
6-2 Using Batch Error Execution Mode 6-11
6-3 Implicit Describe - Select List Is Available as an Attribute of the Statement Handle 6-13
6-4 Explicit Describe - Returning the Select-List Description for Each Column 6-15
6-5 Access on a Scrollable Cursor 6-21
7-1 Handle Allocation and Binding for Each Placeholder in a SQL Statement 7-6
7-2 Defining a PL/SQL Statement to Be Used in OCI 7-7
7-3 Binding the Placeholder and Executing the Statement to Insert a Single Locator 7-11
7-4 Binding the Placeholder and Executing the Statement to Insert an Array of Locators 7-11
7-5 Allowed: Inserting into C1, C2, and L Columns Up to 8000, 8000, and 2000 Byte-Sized Bind Variable Data Values, Respectively 7-14
7-6 Allowed: Inserting into C1 and L Columns up to 2000 and 8000 Byte-Sized Bind Variable Data Values, Respectively 7-14
7-7 Allowed: Updating C1, C2, and L Columns up to 8000, 8000, and 2000 Byte-Sized Bind Variable Data Values, Respectively 7-15

7-8 Allowed: Updating C1, C2, and L Columns up to 2000, 2000, and 8000 Byte-Sized Bind Variable Data Values, Respectively 7-15
7-9 Allowed: Piecewise, Callback, and Array Insert or Update Operations 7-15
7-10 Not Allowed: Inserting More Than 4000 Bytes into Both LOB and LONG Columns Using the Same INSERT Statement 7-15
7-11 Allowed: Inserting into the CT3 LOB Column up to 2000 Byte-Sized Bind Variable Data Values 7-16
7-12 Not Allowed: Binding Any Length Data to a LOB Column in an Insert As Select Operation 7-16
7-13 Defining a Scalar Output Variable Following an Execute and Describe Operation 7-18
7-14 Defining LOBs Before Execution 7-21
7-15 Defining LOBs After Execution 7-21
7-16 Using Multiple Bind and Define Buffers 7-26
7-17 Binding the REF Output Variable in an OCI Application 7-32
7-18 Setting the Client Character Set to OCI_UTF16ID in OCI 7-36
7-19 Insert and Select Operations Using the OCI_ATTR_MAXCHAR_SIZE Attribute 7-41
7-20 Binding and Defining UTF-16 Data 7-42
7-21 Binding the :cursor1 Placeholder to the Statement Handle stm2p as a REF CURSOR 7-44
7-22 Defining a Nested Table (Second Position) as a Statement Handle 7-44
8-1 Initializing the OCI Process in Object Mode 8-4
8-2 Using an Explicit Describe to Retrieve Column Data Types for a Table 8-28
8-3 Describing the Stored Procedure 8-30
8-4 Using an Explicit Describe on a Named Object Type 8-32
8-5 Using an Explicit Describe on a Named Collection Type 8-33
8-6 Using a Parameter Descriptor to Retrieve the Data Types, Column Names, and Character-Length Semantics 8-35
8-7 Checking for Invisible Columns 8-36
9-1 Implementing Read Callback Functions Using OCILobRead2() 9-16
9-2 Implementing Write Callback Functions Using OCILobWrite2() 9-17
9-3 Using Temporary LOBs 9-22
9-4 Prefetching of LOB Data, Length, and Chunk Size 9-26
10-1 Defining the OCI_ATTR_SERVER_GROUP Attribute to Pass the Server Group Name 10-14
10-2 Defining the OCI_ATTR_PROXY_CREDENTIALS Attribute to Specify the Credentials of the Application Server for Client Authentication 10-17
10-3 Defining the OCI_ATTR_DISTINGUISHED_NAME Attribute to Pass the Distinguished Name of the Client 10-17
10-4 Defining the OCI_ATTR_CERTIFICATE Attribute to Pass the Entire X.509 Certificate 10-18
10-5 Defining the OCI_ATTR_INITIAL_CLIENT_ROLES Attribute to Pass the Client Roles 10-18

10-6 Defining the OCI_ATTR_CLIENT_IDENTIFIER Attribute to Pass the End-User Identity 10-19
10-7 Defining the OCI_ATTR_PASSWORD Attribute to Pass the Password for Validation 10-19
10-8 OCI Attributes That Let You Specify the External Name and Initial Privileges of a Client 10-20
10-9 Defining the OCI_ATTR_APPCTX_SIZE Attribute to Initialize the Context Array Size with the Desired Number of Context Attributes 10-23
10-10 Using the OCI_ATTR_APPCTX_LIST Attribute to Get a Handle on the Application Context List Descriptor for the Session 10-23
10-11 Calling OCIParamGet() to Obtain an Individual Descriptor for the i-th Application Context Using the Application Context List Descriptor 10-23
10-12 Defining Session Handle Attributes to Set Externally Initialized Context 10-24
10-13 Using the OCI_ATTR_CALL_TIME Attribute to Get the Elapsed Time of the Last Server Call 10-25
10-14 Using OCISessionBegin() with an Externally Initialized Context 10-26
10-15 Changing the "responsibility" Attribute Value in the CLIENTCONTEXT Namespace 10-29
10-16 Two Ways to Clear Specific Attribute Information in a Client Namespace 10-29
10-17 Clearing All the Context Information in a Specific Client Namespace 10-30
10-18 Calling OCIAttrSet() to Set the OCI_ATTR_EDITION Attribute 10-31
11-1 Example of PL/SQL Fix-Up Callback 11-4
12-1 Event Notification 12-6
12-2 User-Defined Failover Callback Function Definition 12-11
12-3 Failover Callback Registration 12-12
12-4 Failover Callback Unregistration 12-12
12-5 Callback Function That Implements a Failover Strategy 12-13
12-6 Transaction Guard Demo Program 12-17
13-1 Setting QOS Levels, the Notification Grouping Class, Value, and Type, and the Namespace Specific Context 13-7
13-2 Using AQ Grouping Notification Attributes in an OCI Notification Callback 13-13
13-3 Implementing a Publish Subscription Notification 13-15
13-4 Registering for Notification Using Callback Functions 13-17
13-5 LDAP Registration 13-20
13-6 Enqueue Buffered Messaging 13-31
13-7 Dequeue Buffered Messaging 13-31
14-1 Pseudocode That Describes the Overall Processing of a Typical OCI Call 14-4
14-2 Environment Variable Setting for the ORA_OCI_UCBPKG Variable 14-10
14-3 Specifying the pkgNInit() and PkgNEnvCallback() Functions 14-10
14-4 Using pkgNEnvCallback() to Register Entry, Replacement, and Exit Callbacks 14-10
14-5 Registering User Callbacks with the NULL ucbDesc 14-10

14-6 Using the OCIStmtPrepare() Call to Call the Callbacks in Order 14-10
15-1 Optimizing Bind and Define Operations on Statements in the Cache 15-5
15-2 Implicit Fetching of ROWIDs 15-8
15-3 DBMS_SQL RETURN_RESULT Subprogram 15-10
15-4 A PL/SQL Stored Procedure to Implicitly Return Result-Sets (Cursors) to the Client 15-10
15-5 An Anonymous PL/SQL Block to Implicitly Return Result-Sets (Cursors) to the Client 15-11
15-6 Using OCIStmtGetNextResult() to Retrieve and Process the Implicit Results Returned by Either a PL/SQL Stored Procedure or Anonymous Block 15-11
16-1 Calling OCIDBStartup() to Perform a Database Startup Operation 16-2
16-2 Calling OCIDBShutdown() in OCI_DBSHUTDOWN_FINAL Mode 16-3
16-3 Calling OCIDBShutdown() in OCI_DBSHUTDOWN_ABORT Mode 16-4
19-1 SQL Definition of Standalone Objects 19-5
19-2 SQL Definition of Embedded Objects 19-5
19-3 Pinning an Object 19-12
19-4 Manipulating Object Attributes in OCI 19-14
19-5 Using Complex Object Retrieval in OCI 19-25
19-6 C Representations of Types with Their Corresponding NULL Indicator Structures 19-30
19-7 Creating a New Object for an Object View 19-34
20-1 Manipulating an Attribute of Type OCIDate 20-6
20-2 Manipulating an Attribute of Type OCIDateTime 20-10
20-3 Manipulating an Attribute of Type OCINumber 20-12
20-4 Converting Values in OCINumber Format Returned from OCIDescribeAny() Calls to Unsigned Integers 20-14
20-5 Manipulating an Attribute of Type OCIString 20-15
20-6 Manipulating an Attribute of Type OCIRaw 20-17
20-7 Using Collection Data Manipulation Functions 20-20
20-8 Using Multilevel Collection Data Manipulation Functions 20-23
20-9 Using REF Manipulation Functions 20-25
20-10 Using Type Interfaces to Construct Object Types 20-27
20-11 Using Type Interfaces to Construct Collection Types 20-28
20-12 Using Special Construction and Access Calls for Improved Performance 20-31
20-13 Method 1 for a Salary Update: Fetch, Convert, and Assign 20-44
20-14 Method 2 for a Salary Update: Fetch and Assign, No Convert 20-44
20-15 Method 3 for a Salary Update: Direct Fetch 20-44
20-16 Using the SQLT_NTY Bind Call Including OCIBindObject() 20-46
20-17 Using the SQLT_NTY Define Call Including OCIDefineObject() 20-47
21-1 Direct Path Programs Must Include the Header Files 21-4

21-2 Passing the Handle Type to Allocate the Function Context 21-5
21-3 Explicit Allocation of Direct Path Column Array Handle 21-6
21-4 Explicit Allocation of Direct Path Function Column Array Handle 21-6
21-5 Allocating a Direct Path Stream Handle 21-7
21-6 Data Structures Used in Direct Path Loading Examples 21-9
21-7 Contents of the Header File cdemodp.h 21-9
21-8 Use of OCI Direct Path Interfaces 21-12
21-9 Allocating the Column Array and Stream Handles 21-12
21-10 Getting the Number of Rows and Columns 21-13
21-11 Setting Input Data Fields 21-13
21-12 Resetting the Column Array State 21-13
21-13 Resetting the Stream State 21-13
21-14 Converting Data to Stream Format 21-13
21-15 Loading the Stream 21-13
21-16 Finishing the Direct Path Load Operation 21-14
21-17 Freeing the Direct Path Handles 21-14
21-18 Allocating a Child Column Array for a Column Object 21-20
21-19 Allocating a Child Column Array for a SQL String Column 21-23
21-20 Allocating a Child Column Array for a REF Column 21-27
21-21 Allocating the Column Array for the Object Table 21-31
21-22 Specifying Values for the OCI_ATTR_DIRPATH_EXPR_TYPE Attribute 21-37
21-23 Setting a Function Context as a Column Attribute 21-39
21-24 Allocating a Child Column Array for a Function Context 21-42
22-1 Object Type Representation of a Department Row 22-18
22-2 C Representation of a Department Row 22-18
23-1 Initializing and Terminating XML Context with a C API 23-2
24-1 Definition of the Employee Object Type Listed in the Intype File 24-2
24-2 Contents of the Generated Header File demo.h 24-3
24-3 Contents of the demov.c File 24-3
24-4 Invoking OTT from the Command Line 24-5
24-5 Contents of a User-Created Intype File 24-9
24-6 Object Type Definition for Employee 24-10
24-7 OTT-Generated Struct Declarations 24-10
24-8 Object Type Definitions for the OTT Type Mapping Example 24-13
24-9 Various Type Mappings Created by OTT from Object Type Definitions 24-13
24-10 Object Type and Subtype Definitions 24-16
24-11 Contents of the Intype File 24-16

24-12 OTT Generates C Structs for the Types and Null Indicator Structs 24-16
24-13 Contents of an Intype File 24-19
24-14 Contents of the Outtype File After Running OTT 24-19
24-15 Content of an Intype File Named ex2c.typ 24-22
24-16 Invoking OTT and Specifying the Initialization Function 24-22
24-17 Content of an OTT-Generated File Named ex2cv.c 24-22
24-18 Object Type Definition to Demonstrate How OTT Generates Include Files 24-33
24-19 Content of the Intype File 24-33
24-20 Invoking OTT from the Command Line 24-33
24-21 Content of the Header File tott95b.h 24-33
24-22 Content of the Header File tott95a.h 24-34
24-23 Construct to Use to Conditionally Include the Header File tott95b.h 24-34
26-1 Creating a Compound Sharding Key 26-147
26-2 Custom Pool Example 26-150
36-1 Creating a Collection 36-13
36-2 Creating a Document 36-27
F-1 Setting Configuration Parameters and Starting agtctl F-5

# List of Figures

4-1 Basic OCI Program Flow 4-2
4-2 Components of a Service Context 4-7
4-3 Statement Handles 4-8
4-4 Direct Path Handles 4-10
6-1 Steps in Processing SQL Statements 6-2
7-1 Using OCIBindByName() to Associate Placeholders with Program Variables 7-2
7-2 Determining Skip Parameters 7-23
7-3 Performing Piecewise Insert 7-49
7-4 Performing Piecewise Fetch 7-53
8-1 OCIDescribeAny() Table Description 8-2
10-1 Multiple Tightly Coupled Branches 10-5
10-2 Session Operating on Multiple Branches 10-5
11-1 OCI Connection Pooling 11-17
13-1 Publish-Subscribe Model 13-3
19-1 Basic Object Operational Flow 19-7
21-1 Direct Path Loading 21-1
21-2 Inheritance Hierarchy for a Column of Type Person 21-29
22-1 Object Cache 22-3
22-2 Object Graph of person_t Instances 22-20
24-1 Using OTT with OCI 24-20
27-1 Classification of Instances by Type and Lifetime 27-2
35-1 Execution Flow of the OCIXStreamInLCRCallbackSend() Function 35-46
35-2 Execution Flow of the OCIXStreamOutLCRCallbackReceive() Function 35-63
F-1 Multithreaded extproc Agent Architecture F-3

# List of Tables

2-1 Instant Client Shared Libraries for Linux and UNIX 2-2
2-2 Instant Client Shared Libraries for Microsoft Windows 2-2
2-3 Instant Client Light Shared Libraries Linux and UNIX 2-9
2-4 Instant Client Light Shared Libraries for Microsoft Windows 2-9
3-1 ORACLE_HOME Directories and Contents 3-2
3-2 Equivalent OCI Parameter Settings in oraaccess.xml and sqlnet.ora 3-11
3-3 Obsolescent OCI Functions 3-23
3-4 OCI Functions Not Supported 3-24
4-1 OCI Handle Types 4-3
4-2 Descriptor Types 4-12
4-3 OCI Return Codes 4-27
4-4 Return and Error Codes 4-28
4-5 Oracle Reserved Namespaces 4-35
5-1 Internal Oracle Database Data Types 5-4
5-2 External Data Types and Codes 5-8
5-3 VARNUM Examples 5-16
5-4 Format of the DATE Data Type 5-17
5-5 Data Conversions 5-30
5-6 Data Conversions for LOBs 5-31
5-7 Data Conversion for Datetime and Interval Types 5-32
5-8 Data Conversion for External Data Types to Internal Numeric Data Types 5-34
5-9 Data Conversions for Internal to External Numeric Data Types 5-34
5-10 OCITypeCode Values and Data Types 5-35
5-11 OCI_TYPECODE to SQLT Mappings 5-37
6-1 OCI_ATTR_STMT_TYPE Values and Statement Types 6-4
7-1 Information Summary for Bind Types 7-9
8-1 Attributes of All Parameters 8-5
8-2 Attributes of Tables or Views 8-7
8-3 Attributes Specific to Tables 8-8
8-4 Attributes of Procedures or Functions 8-8
8-5 Attributes Specific to Package Subprograms 8-8
8-6 Attributes of Packages 8-9
8-7 Attributes of Types 8-9
8-8 Attributes of Type Attributes 8-11
8-9 Attributes of Type Methods 8-12

8-10 Attributes of Collection Types 8-13
8-11 Attributes of Synonyms 8-14
8-12 Attributes of Sequences 8-15
8-13 Attributes of Columns of Tables or Views 8-16
8-14 Predefined Collation IDs, Their ub4 Values (in parenthesis), and Their SQL Names 8-19
8-15 Attributes of Arguments and Results 8-20
8-16 List Attributes 8-22
8-17 Attributes Specific to Schemas 8-22
8-18 Attributes Specific to Databases 8-23
8-19 Attributes Specific to Rules 8-24
8-20 Attributes Specific to Rule Sets 8-24
8-21 Attributes Specific to Evaluation Contexts 8-25
8-22 Attributes Specific to Table Aliases 8-25
8-23 Attributes Specific to Variable Types 8-25
8-24 Attributes Specific to Name-Value Pair 8-26
9-1 LOB Functions Compatibility and Migration 9-8
10-1 Global Transaction Identifier 10-4
10-2 One-Phase Commit 10-9
10-3 Two-Phase Commit 10-10
10-4 Read-Only Update Fails 10-10
10-5 Read-Only Transaction 10-11
10-6 Initialization and Termination Multithreading Functions 10-39
10-7 Passive Threading Primitives 10-40
10-8 Active Threading Primitives 10-43
12-1 Time and Event 12-13
13-1 Publish-Subscribe Functions 13-12
13-2 AQ Functions 13-25
13-3 Enqueue Parameters 13-25
13-4 Dequeue Parameters 13-25
13-5 Listen Parameters 13-26
13-6 Array Enqueue Parameters 13-26
13-7 Array Dequeue Parameters 13-27
13-8 Agent Parameters 13-27
13-9 Message Properties 13-28
13-10 Enqueue Option Attributes 13-29
13-11 Dequeue Option Attributes 13-29
15-1 Comparison of Some Connection Specific Auto-Tuning Parameters 15-17

19-1 Meta-Attributes of Persistent Objects 19-17
19-2 Set and Check Functions 19-20
19-3 Transient Meta-Attributes 19-20
19-4 Attribute Values for New Objects 19-32
20-1 Function Prefix Examples 20-5
20-2 Binding and Defining Datetime and Interval Data Types 20-8
20-3 Datetime Functions 20-9
20-4 Interval Functions 20-11
20-5 String Functions 20-15
20-6 Raw Functions 20-16
20-7 Collection Functions 20-19
20-8 Collection Scanning Functions 20-19
20-9 Nested Table Functions 20-21
20-10 REF Manipulation Functions 20-24
20-11 Descriptor Objects 20-26
20-12 Data Type Mappings for Binds and Defines 20-40
21-1 Direct Path Context Functions 21-7
21-2 Direct Path Column Array Functions 21-8
22-1 Object Attributes After a Refresh Operation 22-12
22-2 Example of Allocation and Pin Durations 22-17
22-3 Pin, Free, and Unpin Functions 22-22
22-4 Flush and Refresh Functions 22-22
22-5 Mark and Unmark Functions 22-23
22-6 Object Meta-Attributes Functions 22-23
22-7 Other Object Functions 22-23
24-1 Object Data Type Mappings for Object Type Attributes 24-11
25-1 Mode of a Parameter 25-2
25-2 Connect, Authorize, and Initialize Functions 25-3
25-3 Handle and Descriptor Functions 25-46
25-4 Bind, Define, and Describe Functions 25-59
26-1 Statement Functions 26-2
26-2 LOB Functions 26-15
26-3 Advanced Queuing and Publish-Subscribe Functions 26-79
26-4 Direct Path Loading Functions 26-97
26-5 Thread Management Functions 26-110
26-6 Transaction Functions 26-132
26-7 Sharding Functions 26-146

26-8 Miscellaneous Functions 26-152
26-9 OCI Function Codes 26-173
26-10 Continuation of OCI Function Codes from 97 and Higher 26-174
27-1 Type and Lifetime of Instances 27-3
27-2 Return Values of Navigational Functions 27-4
27-3 OCI Navigational Functions Error Codes 27-4
27-4 Flush or Refresh Functions 27-6
27-5 Object Status After Refresh 27-12
27-6 Mark or Unmark Object and Cache Functions 27-13
27-7 Get Object Status Functions 27-20
27-8 Miscellaneous Object Functions 27-26
27-9 Instances Created 27-38
27-10 Pin, Unpin, and Free Functions 27-42
27-11 Type Information Accessor Functions 27-55
28-1 Function Return Values 28-2
28-2 Collection and Iterator Functions 28-3
28-3 Element Pointers 28-8
28-4 Date Functions 28-25
28-5 Error Bits Returned by the valid Parameter for OCIDateCheck() 28-30
28-6 Comparison Results 28-31
28-7 Error Bits Returned by the valid Parameter for OCIDateTimeCheck() 28-40
28-8 Comparison Results Returned by the result Parameter for OCIDateTimeCompare() 28-42
28-9 Error Bits Returned by the valid Parameter for OCIIntervalCheck() 28-61
28-10 Comparison Results Returned by the result Parameter for OCIIntervalCompare() 28-63
28-11 NUMBER Functions 28-74
28-12 Comparison Results Returned by the result Parameter for OCINumberCmp() 28-82
28-13 Values of result 28-103
28-14 Raw Functions 28-111
28-15 Ref Functions 28-117
28-16 String Functions 28-125
28-17 Table Functions 28-131
29-1 External Procedures Functions 29-2
29-2 Memory Services Functions 29-6
29-3 Maintaining Context Functions 29-11
29-4 Parameter Manager Interface Functions 29-15
29-5 File I/O Interface Functions 29-26
29-6 String Formatting Functions 29-35

29-7 Format Modifier Flags 29-38
29-8 Format Codes to Specify How to Format an Argument Written to a String 29-39
30-1 Function Return Values 30-1
30-2 Type Interface Functions 30-2
30-3 Any Data Functions 30-7
30-4 Data Types and Attribute Values 30-10
30-5 Data Types and Attribute Values 30-13
30-6 Any Data Set Functions 30-24
31-1 Function Return Values 31-1
31-2 OCI Locale Functions 31-2
31-3 OCI Locale-Mapping Function 31-8
31-4 OCI String Manipulation Functions 31-10
31-5 OCI Character Classification Functions 31-34
31-6 OCI Character Set Conversion Functions 31-41
31-7 OCI Messaging Functions 31-46
32-1 Function Return Values 32-1
32-2 OCI XML DB Functions 32-2
35-1 Mode of a Parameter 35-5
35-2 OCI XStream Functions 35-5
35-3 Table Column Data Types 35-23
35-4 Required Column List in the First LCR 35-53
35-5 Storage of LOB or LONG Data in the LCR 35-67
36-1 OCI SODA Functions 36-1
A-1 Function Code of the SQL Command Associated with the SQL Statement A-47
B-1 OCI Demonstration Programs B-1
C-1 Server Round-Trips for Relational Operations C-2
C-2 Server Round-Trips for OCILob Calls C-3
C-3 Server Round-Trips for Object and Cache Functions C-5
C-4 Server Round-Trips for Describe Operations C-6
C-5 Server Round-Trips for Data Type Manipulation Functions C-7
C-6 Server Round-Trips for Any Type and Data Functions C-7
C-7 Locally Processed Functions C-8
D-1 ORACLE_HOME Directories and Contents D-2
D-2 Oracle XA Components D-4
D-3 Link Libraries D-5
E-1 Deprecated OCI Functions E-1
E-2 Deprecated Initialize Functions E-2

E-3 Deprecated Statement Functions E-6
E-4 Deprecated LOB Functions E-9
E-5 Characters or Bytes in amtp for OCILobRead() E-16
E-6 Characters or Bytes in amtp for OCILobWrite() E-20
E-7 Characters or Bytes in amtp for OCILobWriteAppend() E-25
E-8 Deprecated Database Advanced Queuing Functions E-27
F-1 Agent Control Utility (agtctl) Commands F-5
F-2 Configuration Parameters for agtctl F-10

# Preface

Oracle Call Interface (OCI) is an application programming interface (API) that lets applications written in C or C++ interact with Oracle Database. OCI gives your programs the capability to perform the full range of database operations that are possible with Oracle Database, 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 Database environment. This comprehensive treatment of OCI is also 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 assume knowledge of the basic concepts of object-oriented programming.

**See Also:**

- *Oracle Database SQL Language Reference*and*Oracle Database Administrator's Guide* for information about SQL
- *Oracle Database Concepts*
- *Oracle Database New Features Guide* for information about the differences between the Standard Edition and the Enterprise Edition and all the features and options that are available to you
- *Oracle C++ Call Interface Programmer's Guide* for more information about OCI functionality for C++ that enables programmers to manipulate database objects of user-defined types as C++ objects

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

### Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

## 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. See *Oracle Database Sample Schemas* for information about how these schemas were created and how you can use them.

To download free release notes, installation documentation, white papers, or other collateral, 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/technetwork/community/join/overview/`

If you 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/technetwork/indexes/documentation/`

*Oracle Call Interface Programmer's Guide* does not contain all information that describes the features and functionality of OCI in the Oracle Database Standard Edition and Enterprise Edition products. Explore the following documents for additional information about OCI.

- *Oracle Database Data Cartridge Developer's Guide* provides information about cartridge services and OCI calls pertaining to development of data cartridges.
- *Oracle Database Globalization Support Guide* explains OCI calls pertaining to NLS settings and globalization support.
- *Oracle Database Advanced Queuing User's Guide* supplies information about OCI calls pertaining to Advanced Queuing.
- *Oracle Database Development Guide* explains how to use OCI with the XA library.
- *Oracle Database SecureFiles and Large Objects Developer's Guide* provides information about using OCI calls to manipulate LOBs, including code examples.
- *Oracle Database Object-Relational Developer's Guide* offers a detailed explanation of object types.

For additional information about Oracle Database, consult the following documents:

- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database New Features Guide*
- *Oracle Database Concepts*
- *Oracle Database Reference*
- *Oracle Database Error Messages Reference*

# 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. |

# Changes in This Release for Oracle Call Interface Programmer's Guide

This preface contains:

- Changes in Oracle Database release 18*c*, version 18.1

## Changes in Oracle Database Release 19c Version 19.1

The following are changes in this book for Oracle Database release 19c, version 19.1.

### New Features

New features for Oracle Database release 19c, version 19.1

The following new OCI SODA functions are included in this release:

- `OCISodaBulkInsert()`, `OCISodaBulkInsertAndGet()`, `OCISodaBulkInsertAndGetWithCtnt()`, `OCISodaBulkInsertWithCtnt()`, `OCISodaDataGuideGet()`, `OCISodaDocCount()`, `OCISodaDocCountWithFilter()`, `OCISodaDocGetNext()`, `OCISodaFind()`, `OCISodaFindOne()`, `OCISodaIndexCreate()`, `OCISodaIndexDrop()`, `OCISodaRemove()`, `OCISodaReplOne()`, `OCISodaReplOneAndGet()`, `OCISodaReplOneAndGetWithCtnt()`, `OCISodaReplOneWithCtnt()`, and `OCISodaOperKeysSet()`.

  See OCI SODA Functions.

### Other Changes

The following additional changes were made in this release:

- All occurrences of "Streams Advanced Queuing" in the document are replaced with "Database Advanced Queuing".

## Changes in Oracle Database Release 18c Version 18.1

Changes in Oracle Database release 18c, version 18.1.

The following are changes in the Oracle Call Interface (OCI):

- New Features

### New Features

New features for Oracle Database release 18c, version 18.1

The following features are new in this release:

- OCI support for Simple Oracle Document Access (SODA)

  SODA for C is a C API that is part of Oracle Call Interface (OCI). It implements Simple Oracle Document Access (SODA). You can use it to perform create, read (retrieve), update, and delete (CRUD) operations on documents of any kind, and you can use it to query JSON documents. You compile programs that use SODA for C the same way you compile other OCI programs. SODA is a set of NoSQL-style APIs that let you create and store collections of documents in Oracle Database, retrieve them, and query them, without needing to know Structured Query Language (SQL) or how the data in the documents is stored in the database. Oracle Database supports storing and querying JSON data. To access this functionality, you need structured query language (SQL) with special JSON SQL operators. SODA for C hides the complexities of SQL/JSON programming.

  See the following for more information: SODA for C OverviewOCI SODA Functions, Handles, SODA Collection Handle Attributes , SODA Document Handle Attributes , SODA Operation Options Handle Attributes, and SODA Output Options Handle Attributes.

- OCI support for operation level timeout or fine grained timers for operations

  New service context attribute `OCI_ATTR_CALL_TIMEOUT` — For setting millisecond timeout values using the `OCIAttrSet()` function. Allows applications to specify a timeout value for network calls to the database.

  See OCI_ATTR_CALL_TIMEOUT for more information.

- OCI support for Application Continuity in an HA infrastructure includes:
  - Support is added for OCI dynamic binds and defines for numeric, character, and date/time types. This means the following OCI APIs are extended to support Application Continuity: `OCIBindDynamic()` and `OCIDefineDynamic()`.
  - Support is added for binding and defining objects. This means the following OCI APIs are extended to support Application Continuity: `OCIBindObject()`, `OCIDefineObject()`, and `OCITypeByName()`.
  - During execution of LOB calls, Application Continuity now supports the handling of connection failure by restarting LOB calls that were interrupted by an outage.
  - OCI now supports the new Application Continuity `FAILOVER_TYPE` of `AUTO`, which only attempts to fail over if the session state is known to be restorable at the explicit request boundary.

  See About Added Support for Application Continuity for more information.

- OCI support for collations for bind variables

  Using `OCIAttrSet()`, the `OCI_ATTR_COLLATION_ID` attribute can be set on a bind variable handle to any of the supported collation IDs. The collation set with `OCI_ATTR_COLLATION_ID` is used as the derived collation of the corresponding placeholder expression (host variable reference) in the SQL statement.

  See OCI_ATTR_COLLATION_ID for more information.

- OCI support for Oracle Connection Manager in Traffic Director Mode

  Oracle Connection Manager in Traffic Director Mode is a proxy that is placed between the database clients and the database instances. Supported OCI clients from Oracle Database 11*g* Release 2 (11.2) and later can connect to it. Oracle Connection Manager in Traffic Director Mode provides improved high availability

(HA) for planned and unplanned database server outages, connection multiplexing support, and load balancing.

Oracle Connection Manager in Traffic Director Mode can use the OCI client `oraaccess.xml` configuration file to configure proxy resident connection pools for one or more services that provide a proxy between the client and database instances. This feature provides improved high availability and performance for both planned and unplanned outages.

See About Using Oracle Connection Manager in Traffic Director Mode and About OCI Session Pool Configuration in oraaccess.xml for more information.

- New LOB descriptor attribute: `OCI_ATTR_LOB_TYPE` — Used to get the LOB type from the LOB descriptor.

  See OCI_ATTR_LOB_TYPE for more information.

- New LOB locator attribute: `OCI_ATTR_LOB_REMOTE` — Determines whether the LOB locator belongs to a local database table or a remote database table.

  See OCI_ATTR_LOB_REMOTE for more information.

- `OCI_ATTR_STMT_TYPE` — has one additional statement attribute types:
  - `OCI_STMT_MERGE` is added beginning Oracle Database release 18c, version 18.1.

  See About Preparing Statements and OCI_ATTR_STMT_TYPE for more information about these statement attribute types.

- New user session handle attribute: `OCI_ATTR_AUTOTUNING_ENABLED` — checks if the client user session has OCI auto tuning enabled on OCI client statement caching.

  See OCI_ATTR_AUTOTUNING_ENABLED for more information.

- Instant Client Basic and Basic Light ZIP files on Linux, Unix, and macOS now have the `libclntsh` and `libocci` symbolic links precreated.

  See the Note in Install from ZIP Files, from RPMs, and from Oracle Universal Installer for more information.

- Instant Client Basic and Basic Light Zips and RPMs now precreate a `network/admin` directory, where optional files such as, `tnsnames.ora`, `sqlnet.ora`, and `oraaccess.xml` can be placed.

  See the Note in SDK for Oracle Instant Client for more information.

- New session pool handle attribute: `OCI_ATTR_SPOOL_MAX_USE_SESSION` — Sets the maximum number of times one session can be checked out of the session pool, after which the session is automatically destroyed.

  See OCI_ATTR_SPOOL_MAX_USE_SESSION for more information.

- Two new column property flags for the `OCI_ATTR_COL_PROPERTIES` attibute:
  - `OCI_ATTR_COL_PROPERTY_IS_LPART` — Indicates whether this is an implicitly generated logical partitioning column for `container_map` enabled objects.
  - `OCI_ATTR_COL_PROPERTY_IS_CONID` — Indicates whether this is a `CON_ID` column implicitly generated by `CONTAINERS()` or is an `ORIGIN_CON_ID` column implicitly generated for Extended Data Link.

  See Column Attributes for more information.

- Changes in server version macros

  Because the components of the database version changed beginning with release 18c, version 18.1, there are new client macros provided for extracting each of the components. As a result, the signature for the `OCIClientVersion()` call is changed and there is a new `OCIServerRelease2()` call. The macros are named according to the new release naming scheme beginning with version 18.1.

  See OCIClientVersion() and OCIServerRelease2() for more information.

## Deprecated Features

Deprecated features in Oracle Database release 18c, version 18.1.

The following features are deprecated in this release, and may be desupported in a future releases:

- `OCI_DTYPE_RSET` — Result set descriptor.

## 1

# OCI: Introduction

This chapter contains these topics:

- Overview of OCI
- Building an OCI Application
- Alternatives to OCI
- SQL Statements
- Procedural and Nonprocedural Elements
- Object Support
- Simple Oracle Document Access (SODA)
- Encapsulated Interfaces
- User Authentication and Password Management
- Features to Improve Application Performance and Scalability
- Oracle Database Advanced Queuing
- XA Library Support

## 1.1 Overview of OCI

Oracle Call Interface (OCI) is an application programming interface (API) that lets you create applications that use function calls to access an Oracle database and control all phases of SQL statement execution and data access.

OCI supports the data types, calling conventions, syntax, and semantics of C and C++.

OCI provides:

- High 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 Database objects
- Access to external databases, such as Oracle TimesTen In-Memory Database and Oracle In-Memory Database Cache. See *Oracle TimesTen In-Memory Database C Developers Guide*.
- Applications that support an increasing number of users and requests without additional hardware investments
- Ways to manipulate data and schemas in an Oracle Database using the C programming language and 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 run time.

- 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
- 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
  - 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
  - Data type 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

**See Also:**

- Oracle Call Interface
- Related Documents

## 1.2 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.

OCI supports most popular 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 Database system-specific documentation and the installation guide for more information about compiling and linking an OCI application for your operating system.

**See Also:**

- Building and Configuring OCI Applications
- SDK for Oracle Instant Client

## 1.3 Alternatives to OCI

Some alternatives to using the Oracle Call Interface (OCI) include:

- Oracle Database Programming Interface for C (ODPI-C)
- Oracle C++ Call Interface (OCCI)
- Oracle Pro*C/C++ Precompiler
- Oracle ODBC Driver

**Oracle Database Programming Interface for C (ODPI-C)**

ODPI-C is an open source library of C code that simplifies the use of common Oracle Call Interface (OCI) features for Oracle Database drivers and user applications. ODPI-C sits on top of OCI and requires Oracle client libraries. ODPI-C:

- Favors ease of use aimed at driver writers where niche special-case OCI features are not needed.
- Provides a faster implementation of drivers with considerably less code. Oracle features can be exposed to users rapidly and in a consistent way.
- Provides simpler memory management for binding variables and fetching.
- Automatically converts binding and 'defining' (for fetches) to "native" C types so that additional calls do not need to be made. This is beneficial particularly for

numbers and dates. The ability to retrieve LONG and LOB columns as strings and buffers is an advantage as well.

- Provides a "safer" API in that resource handles are validated. Casts are not needed. A reference counting mechanism adds resiliency by stopping applications from destroying in-use OCI resources.
- Provides an API that simplifies connection and resource management. For example, it automatically does session pool pinging to provide better High Availability.
- Provides an alternative programming experience from OCI that uses a multiple getter and setter model for handling attributes.
- Provides a sample Makefile that builds ODPI-C as a shared library. Or, the ODPI-C source code can be included in your project and built as you would build an OCI application.

> **See Also:**
>
> ODPI-C Home Page, for a list of ODPI-C supported features and a list of references including to its home page, code location on github, and documentation

**Oracle C++ Call Interface (OCCI)**

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

> **See Also:**
>
> *Oracle C++ Call Interface Programmer's Guide*

**Oracle Pro*C/C++ Precompiler**

Oracle Pro*C/C++ Precompiler is a programming tool that enables the user to embed SQL statements in a high-level source program. The precompiler accepts the source program as input, translates the embedded SQL statements into standard Oracle runtime library calls, and generates a modified source program that you can compile, link, and execute in the usual way.

> **See Also:**
>
> *Pro*C/C++ Programmer's Guide*

**Oracle ODBC Driver**

The Oracle ODBC Driver enables ODBC applications on Microsoft Windows, as well as UNIX platforms like Linux, Solaris, IBM Advanced Interactive eXecutive (AIX), HP-UX Itaniutm, and IBM Linux on Platform z read and write access to Oracle® databases through the ODBC interface using Oracle Net Services software.

> **See Also:**
>
> *Oracle Database Development Guide* for more information about the Oracle ODBC Driver

# 1.4 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 Database recognizes several types of SQL statements:

- Data Definition Language (DDL)
- Control Statements
  - Transaction Control
  - Session Control
  - System Control
- Queries
- Data Manipulation Language (DML)
- PL/SQL
- Embedded SQL
- Special OCI Terms for SQL

> **See Also:**
>
> Using SQL Statements in OCI

## 1.4.1 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, as in the following series of statements that create 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;
```

## 1.4.2 Control Statements

OCI applications treat transaction control, session control, and system control statements as if they were DML statements.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about these types of statements

## 1.4.3 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:**
>
> About Binding Placeholders in OCI for more information about input bind variables

DML statements also allow you to work with objects in the Oracle database, 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','987-65-4320','146 Winfield Street'));
```

## 1.4.4 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 is to be supplied by the application.

When processing a query, an OCI application also must define output variables to receive the returned results. In the preceding statement, you must define an output variable to receive any `name` values returned from the query.

**See Also:**

- Overview of Binding in OCI for more information about input bind variables
- Overview of Defining in OCI for information about defining output variables
- Using SQL Statements in OCI for detailed information about how SQL statements are processed in an OCI program

## 1.4.5 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 some 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 Database stored procedures and stored functions
- Combine procedural control statements with several SQL statements, so that they are executed as a unit
- Access special PL/SQL features such as records, tables, cursor FOR loops, and exception handling
- Use cursor variables
- Take advantage of implicit result set capability that allows reuse of existing stored procedure designs that return implicit result sets
- Access and manipulate objects in an Oracle database

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 the database when the statement is processed. These placeholders must be bound to C language variables in your program.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for information about coding PL/SQL blocks
- About Binding Placeholders in PL/SQL for information about working with placeholders in PL/SQL

## 1.4.6 Embedded SQL

OCI processes SQL statements as text strings that an application passes to the database 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*

### 1.4.7 Special OCI Terms for SQL

This guide uses special terms to refer to the different parts of a SQL statement.

For example, consider the following SQL statement:

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

It 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 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.

**See Also:**

- Using SQL Statements in OCI for more information about these terms and operations

## 1.5 Procedural and Nonprocedural Elements

OCI enables you to develop scalable, multithreaded applications in a multitier architecture that combines the nonprocedural data access power of structured query language (SQL) with the procedural capabilities of C and C++.

- In a nonprocedural language program, the set of data to be operated on is specified, but what operations are to be performed, or how the operations are to be conducted, is not specified. The nonprocedural 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, that 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 nonprocedural 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. For example, an OCI program can run a query against an Oracle database. The query 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 is to 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 a database.

## 1.6 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`—to represent a person's identifying characteristics.

The object type definition serves as the basis for creating objects that represent instances of the object type by using the object type as a structural definition, you could create a `person` object with the attribute values 'John', 'Bonivento', and '30'. Object types may also contain *methods*—programmatic functions that represent the behavior of that object type.

OCI provides a comprehensive application programming interface for programmers seeking to use Oracle Database object capabilities.

These capabilities include:

- 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 to synchronize object and relational functionality

OCI object features can be divided into the following major categories:

- Client-Side Object Cache
- Associative and Navigational Interfaces to access and manipulate objects
- OCI Runtime Environment for Objects
- Type Management: Mapping and Manipulation Functions to access information about object types and control data attributes of Oracle types
- Object Type Translator (OTT) utility, for mapping internal Oracle Database schema information to client-side language bind variables

> **See Also:**
>
> - Encapsulated Interfaces which describes additional OCI calls that are provided to support manipulation of objects after they have been accessed by SQL statements
> - *Oracle Database Concepts*
> - *Oracle Database Object-Relational Developer's Guide*

## 1.6.1 Client-Side Object Cache

The object cache is a client-side memory buffer that provides lookup and memory management support for objects.

The object cache 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. When multiple applications run against the same server, each has its own object cache. The cache tracks the objects that 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 features 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

## 1.6.2 Associative and Navigational Interfaces

What are the different types of interfaces OCI applications can use to access objects?

Applications using OCI can access objects in an Oracle database 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 `REF`s

OCI provides a set of functions with extensions to support object manipulation using SQL `SELECT`, `INSERT`, and `UPDATE` statements. To access Oracle Database 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 after they have been fetched into the client-side cache by traversing the corresponding smart pointers or `REF`s. 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

## 1.6.3 OCI Runtime Environment for Objects

OCI provides functions for objects to manage how Oracle Database objects are used on the client side.

These functions provide for:

- Connecting to an Oracle database server 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 database
- Associatively accessing objects through SQL
- Describing PL/SQL procedures or functions whose parameters or results are Oracle types

## 1.6.4 Type Management: Mapping and Manipulation Functions

OCI provides two sets of functions to work with Oracle Database objects.

- Type Mapping functions allow applications to map attributes of an Oracle schema represented in the server as internal Oracle data types 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.

### 1.6.5 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 that contains metadata information about Oracle schema objects. It generates an `outtype` file and the 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 is beneficial because it:

- 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 Database 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 Database, OTT can only generate C structures that can either be used with OCI programs or with the Pro*C/C++ precompiler programs.

## 1.7 Simple Oracle Document Access (SODA)

SODA for C is a C API that is part of Oracle Call Interface (OCI).

SODA for C implements Simple Oracle Document Access (SODA). You can use it to perform create, read (retrieve), update, and delete (CRUD) operations on documents of any kind, and you can use it to query JSON documents. You compile programs that use SODA for C the same way you compile other OCI programs. SODA is a set of NoSQL-style APIs that let you create and store collections of documents in Oracle Database, retrieve them, and query them, without needing to know Structured Query Language (SQL) or how the data in the documents is stored in the database. Oracle Database supports storing and querying JSON data. To access this functionality, you need structured query language (SQL) with special JSON SQL operators. SODA for C hides the complexities of SQL/JSON programming.

**Related Topics**

- Oracle Database Introduction to Simple Oracle Document Access (SODA)
- Oracle Database SODA for C Developers Guide
- OCI SODA Functions
  The following table lists the OCI SODA functions that are described in this chapter.

- Handles
  Almost every OCI call includes in its parameter list one or more handles.
- SODA Collection Handle Attributes
  Lists and describes the OCI SODA collection handle attributes.
- SODA Document Handle Attributes
- SODA Operation Options Handle Attributes
  Lists and describes the OCI SODA Operation Options handle attributes.
- SODA Output Options Handle Attributes
  Lists and describes the OCI SODA Output Options handle attributes. This handle is used to return the number of documents processed by a bulk operation. Currently, it is returned only by bulk insert methods.

## 1.8 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 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 a handle by using accessor functions.

The OCI library manages a hierarchy of handles. Encapsulating the OCI interfaces with these handles has several benefits to the application developer, including:

- Reduction of server-side state information that must 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
- Allows changes to be made to the underlying structure without affecting applications

## 1.9 User Authentication and Password Management

OCI provides application developers with user authentication and password management.

This is supported in several ways:

- OCI enables a single OCI application to authenticate and maintain multiple users.
- OCI enables 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 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 (the session created when a

user logs in to an Oracle database) from the user sessions (all other sessions created by a user)

Privileged connections, such as `SYSDBA`, `SYSOPER`, proxy authentication, external authentication, and others, are also supported.

## 1.10 Features to Improve Application Performance and Scalability

OCI provides several feature extensions 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 must be retained on the server side. Some of these features include:

- Statement caching to improve performance by caching executable statements that are used repeatedly
- Client result caching to limit the number of round trips to the database server
- Implicit prefetching of `SELECT` statement result sets to eliminate the describe round-trip, reduce round-trips, and reduce memory usage
- Elimination of open and close cursor round-trips
- 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 transaction processing (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 by 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

## 1.11 Oracle Database Advanced Queuing

OCI provides an interface to Oracle Database Advanced Queuing (Database AQ) feature.

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

**See Also:**

OCI and Database Advanced Queuing

## 1.12 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 `$ORACLE_HOME/rdbms/public`. Users of the `demo_rdbms.mk` file on Linux or UNIX are not affected because this make file includes the `$ORACLE_HOME/rdbms/public` directory.

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

**See Also:**

- Oracle XA Library for more information about Windows and XA applications
- *Oracle Database Development Guide* for information about developing applications with Oracle XA

## 2

# Oracle Instant Client and Oracle Instant Client Light

This chapter contains the following topics:

- About Oracle Instant Client
- Install from ZIP Files, from RPMs, and from Oracle Universal Installer
- Environment Variables for Oracle Instant Client
- Database Connection Strings for Oracle Instant Client
- SDK for Oracle Instant Client
- About Oracle Instant Client Light
- About Patching Oracle Instant Client Shared Libraries on Linux or UNIX
- Regeneration of Data Shared Library and Zip and RPM Files

## 2.1 About Oracle Instant Client

Oracle Instant Client provides the necessary Oracle Database client-side files to create and run OCI, OCCI, ODBC, and JDBC OCI applications. Useful command-line utilities including SQL*Plus, SQL*Loader and Oracle Data Pump are also available. Oracle Instant Client simplifies the deployment of applications by eliminating the need for an Oracle home on the client machines.

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. A minimal install of Instant Client shared libraries can occupy less than 10% of the space of a full client-side installation. Finally, the Instant Client is easy to install.

Why use Instant Client?

- Installation involves unzipping a few files, or installing RPM packages.
- The Oracle Database 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 using Instant Client.
- It is simple for independent software vendors to package applications.

A README file is included with the Instant Client installation. It describes the version, date and time, and the operating system the Instant Client was generated on.

OCI applications, by default, look for the OCI Data Shared Library, `libociei.so` (or `Oraociei18.1.dll` on Windows) in the runtime library search path (for example `LD_LIBRARY_PATH` on Linux, or `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 (see Table 2-3 and Table 2-4), `libociicus.so` (or `Oraociicus18.1.dll` on Windows). If the Instant Client Light library is found, then the application operates in the Instant Client Light mode. Otherwise, a full installation based on Oracle home is assumed.

For Instant Client, the following tables shows the Oracle Database client-side files required to deploy an OCI application for Linux and UNIX and Microsoft Windows. Other libraries and utilities get installed, but are not needed for OCI runtime use. For example, you can delete files such as `jdbc.jar`.

**Table 2-1 Instant Client Shared Libraries for Linux and UNIX**

| Linux and UNIX | Description for Linux and UNIX |
|---|---|
| `libclntsh.so.18.1`<br>`libclntshcore.so.18.1` | Client Code Library |
| `libociei.so`[1] | OCI Instant Client Data Shared Library |
| `libnnz18.so` | Security Library |
| `libons.so` | Oracle Notification Service (ONS) library |
| `libmql1.so` | Light Weight IPC Library |
| `libipc1.so` | Message Queue Light Library |

[1] The libraries libclntsh.so.18.1, libclntshcore.so.18.1, and libociei.so must reside in the same directory in order to use Instant Client.

**Table 2-2 Instant Client Shared Libraries for Microsoft Windows**

| Microsoft Windows | Description for Microsoft Windows |
|---|---|
| `oci.dll` | Forwarding functions that applications link with |
| `oraociei18.dll` | Data and code |
| `orannzsbb18.dll` | Security Library |
| `oraons.dll` | ONS library used by OCI internally |
| `oci.sym`<br>`oraociei18.sym`<br>`orannzsbb18.sym` | Symbol tables |

Oracle Database 18*c* library names are used in the table and are current for Oracle Database release 18c, version 18.1.

On Microsoft Windows, a `.sym` file is provided for each dynamic-link library (DLL). When the `.sym` file is present in the same location as the DLL, a stack trace with function names is generated when a failure occurs in OCI on Microsoft Windows.

In general, all OCI functionality is available to an application being run using the Instant Client, except that the Instant Client is for client-side operation only. Therefore, server-side external procedures cannot use Instant Client libraries.

For development you can also use the Instant Client SDK.

**See Also:**

- SDK for Oracle Instant Client
- Fault Diagnosability in OCI

## 2.2 Install from ZIP Files, from RPMs, and from Oracle Universal Installer

Oracle Instant Client can be installed in several ways.

The Instant Client libraries can be installed by either choosing the Instant Client option from Oracle Universal Installer or by downloading and installing either the ZIP files or RPMs from the OCI page on the Oracle Technology Network website: Oracle Instant Client

**To Download and Install the Instant Client Libraries from the Oracle Technology Network Website**

The following steps are for the Linux, Unix, and Windows environments.

1. From the **Oracle Technology Network**, download and unzip the required ZIP packages from OTN to an empty directory, such as `instantclient_18_1`, for Oracle Database release 18c, version 18.1. All Instant Client ZIP installs require the Basic or Basic Light package.
2. Or, to install the RPM files, follow these instructions:
   a. Become root by running `su` and entering the super-user password.
   b. Change to the directory in which you want to install.
   c. Uninstall any earlier installations.

      ```
      # rpm -e package-name
      ```

   d. Install each package with yum. Yum will automatically install required dependencies, such as `libaio`.

      ```
      # yum remove package-name
      # yum install package-name.rpm
      ```

      Or, install each package with rpm.

      ```
      # rpm -ivh package-name.rpm
      ```

   e. Delete the `.rpm` files if you want to save disk space.
   f. Exit the root shell. No need to reboot.
3. Set the operating system shared library path environment variable to the directory from Step 1. For example, with ZIP files on Linux or UNIX, set `LD_LIBRARY_PATH` to `instantclient_18_1`.

   ```
   export LD_LIBRARY_PATH=/opt/oracle/instantclient_18_1:$LD_LIBRARY_PATH
   ```

With Linux RPMs, set `LD_LIBRARY_PATH` to:

```
export LD_LIBRARY_PATH=/usr/lib/oracle/18.1/client64/
lib:$LD_LIBRARY_PATH
```

On Windows, set `PATH` to the `instantclient_18_1` directory. For example, on Windows 7, update PATH in Control Panel -> System -> Advanced System Settings -> Advanced -> Environment Variables -> System Variables -> PATH.

4. If necessary, set the `NLS_LANG` environment variable to specify the language and territory used by the client application and database connections opened by the application, and the client's character set, which is the character set for data entered or displayed by a client program. `NLS_LANG` is set as an environment variable on UNIX platforms and is set in the registry on Windows platforms. See *Oracle Database Globalization Support Guide* for more information on setting the `NLS_LANG` environment variable.
5. If you intend to co-locate optional Oracle client configuration files such as `tnsnames.ora`, `sqlnet.ora`, `ldap.ora`, or `oraaccess.xml` with Instant Client, then put them in the `network/admin` subdirectory of Instant Client. This is the default Oracle client configuration directory for applications linked with Instant Client. Alternatively, Oracle client configuration files can be put in another, accessible directory. Then set the environment variable TNS_ADMIN to that directory name.

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

The OCI application uses the Instant Client when the OCI shared libraries are accessible through the operating system Library Path variable. In this way, there is no dependency on the Oracle home and none of the other code and data files provided in the Oracle home are needed by OCI.

### Install the Instant Client from the Oracle Universal Installer

For **Oracle Universal Installer**, invoke the Oracle Universal Installer and select the Instant Client option and install the Instant Client shared libraries to an empty directory, such as `instantclient_18_1`, for Oracle Database release 18c, version 18.1.

See Steps 3 through 5 in To Download and Install the Instant Client Libraries from the Oracle Technology Network Website about setting environment variables.

If you did a complete client installation (by choosing the `Admin` option in Oracle Universal Installer), 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.18.1`, `libclntshcore.so.18.1`, and `libnnz18.so` are in `$ORACLE_HOME/lib`

On Windows:

`oraociei18.dll` library is in `ORACLE_HOME\instantclient`

`oci.dll`, `ociw32.dll`, and `orannzsbb18.dll` are in `ORACLE_HOME\bin`

To enable running the OCI application using Instant Client, copy the preceding libraries to a different directory and set the operating system shared library path to locate this directory.

**Note:**

All the libraries must be copied from the same Oracle home and must be placed in the same directory. Co-location of symlinks to Instant Client libraries is not a substitute for physical co-location of the libraries.

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 performed on the same system, then you should not have `ORACLE_HOME/lib` and the Instant Client directory on the operating system Library Path simultaneously regardless of the order in which they appear on the Library Path. That is, either the `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, but not both.

To enable other capabilities such as OCCI and JDBC OCI, you must copy a few additional files. To enable OCCI, you must install the OCCI Library (`libocci.so.18.1` on Linux or UNIX and `oraocci18.dll` on Windows) in the Instant Client directory. For the JDBC OCI driver, in addition to the three OCI shared libraries, you must also download OCI JDBC Library (for example `libocijdbc18.so` on Linux or UNIX and `ocijdbc18.dll` on Windows). Place all libraries in the Instant Client directory.

**Note:**

On hybrid platforms, such as Sparc64, to operate the JDBC OCI driver in the Instant Client mode, copy the `libociei.so` library from the `ORACLE_HOME/instantclient32` directory to the Instant Client directory. Copy all other Sparc64 libraries needed for the JDBC OCI Instant Client from the `ORACLE_HOME/lib32` directory to the Instant Client directory.

## 2.3 Environment Variables for Oracle 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 affect OCI. OCI always obtains 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. 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 must reside in `ORACLE_HOME/lib` (`ORACLE_HOME\bin` on Windows). Set `ORACLE_HOME` in this case.

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

Beginning with Oracle Database 12*c* Release 2 (12.2), when using Instant Client libraries, the Instant Client can use the `ORA_TZFILE` environment variable to read the time zone file from the file system when this environment variable is set. Otherwise, if the `ORA_TZFILE` variable is not set, then the larger, default, `timezlrg_n.dat` file from the Data Shared Library is used. If the smaller `timezone_n.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. The file must be copied to subdirectory `oracore/zoneinfo` for (UNIX/LINUX) or `oracore\zoneinfo` for (Microsoft Windows) in the instant client directory. You can use the `genezi` utility with the `-v` option to verify that the time zone file and location is valid or run SQL*Plus.

On Linux or UNIX:

```
export ORA_TZFILE=timezone_n.dat
```

On Windows:

```
set ORA_TZFILE=timezone_n.dat
```

In these examples, *n* is the time zone data file version number.

Be sure to create subdirectory `oracore/zoneinfo` for (UNIX/LINUX) or subdirectory `oracore\zoneinfo` for (Microsoft Windows) where `oracoei` or `oraociicus` DLL's are located (typically in the `instantclient_12_2` directory) and copy the `timezone_n.dat` file to this directory.

To determine the versions of small and large time zone files that are packaged in the Instant Client Data Shared Library, enter the following command to run the `genezi` utility:

```
genezi -v
```

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 Database releases.

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

## 2.4 Database Connection Strings for Oracle Instant Client

All Oracle Net naming methods that do not require use of `ORACLE_BASE_HOME`, or `ORACLE_HOME` (to locate configuration files such as `tnsnames.ora`, `sqlnet.ora`, or `oraaccess.xml`) work in the Instant Client mode.

If you intend to co-locate optional Oracle configuration files such as `tnsnames.ora`, `sqlnet.ora`, `ldap.ora`, or `oraaccess.xml` with Instant Client, note that beginning with Oracle Database release 18c, version 18.1 Instant Client Basic and Basic Light Zips and RPMs precreate a `network/admin` directory, where these files can be placed. In earlier releases, users had to create the `network/admin` directory themselves.

See Database Connection Strings for complete information about database connection strings.

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

This section includes the following topic: Examples of Oracle Database Connection String Connect Identifiers.

## 2.5 SDK for Oracle Instant Client

The software development kit (SDK) is a set of development tools that allows the creation of applications can be downloaded from the Instant Client link URL on the Oracle Technology Network website:

Oracle Instant Client

- 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_18_1` directory must be in the runtime library search path, for example `LD_LIBRARY_PATH` before linking the application. The OCI and OCCI programs require the presence of `libclntsh.so` and `libocci.so` symbolic links in the `instantclient_18_1` directory. Beginning with Oracle Database 18c, version 18.1, Instant Client Basic and Basic Light ZIP files on Linux, Unix, and macOS now have the `libclntsh` and `libocci` symbolic links precreated.
- The SDK also contains the Object Type Translator (OTT) utility and its classes to generate the application header files.

## 2.6 About Oracle Instant Client Light

The Instant Client Light version of Instant Client further reduces the disk space requirements of the client installation.

The size of the library has been reduced by removing error message files for languages other than English and leaving only a few supported character set definitions out of around 250.

This Instant Client Light version is geared toward applications that use either US7ASCII, WE8DEC, WE8ISO8859P1, WE8MSWIN1252, or a Unicode character set. There is no restriction on the `LANGUAGE` and the `TERRITORY` fields of the `NLS_LANG` setting, so the Instant Client Light operates with any language and territory settings. Because only English error messages are provided with the Instant Client Light, error messages generated on the client side, such as Net connection errors, are always reported in English, even if `NLS_LANG` is set to a language other than `AMERICAN`. Error messages generated by the database side, such as syntax errors in SQL statements, are in the selected language provided the appropriate translated message files are installed in the Oracle home of the database instance.

This section includes the following topics:

- Globalization Settings
- Libraries for Oracle Instant Client Light
- Install from ZIP Files, from RPMs, and from Oracle Universal Installer

## 2.6.1 Globalization Settings

Instant Client Light supports the following client character sets:

Single-byte

- US7ASCII
- WE8DEC
- WE8MSWIN1252
- WE8ISO8859P1

Unicode

- UTF8
- AL16UTF16
- AL32UTF8

Instant Client Light can connect to databases having one of these database character sets:

- US7ASCII
- WE8DEC
- WE8MSWIN1252
- WE8ISO8859P1
- WE8EBCDIC37C
- WE8EBCDIC1047
- UTF8
- AL32UTF8

Instant Client Light returns an error if a character set other than those in the preceding lists is used as the client or database character set.

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 National Language Support (NLS) settings

## 2.6.2 Libraries for Oracle Instant Client Light

OCI applications, by default, look for the OCI Data Shared Library, `libociei.so` (or `Oraociei18.1.dll` on Windows) in the runtime library search path (for example `LD_LIBRARY_PATH` on Linux, or `PATH` on Windows) to determine if the application should operate in the Instant Client mode.

If the OCI Data Shared Library is not found, then OCI tries to load the Instant Client Light Data Shared Library (see Table 2-3 and Table 2-4), `libociicus.so` (or `Oraociicus18.1.dll` on Windows). If the Instant Client Light library is found, then the application operates in the Instant Client Light mode. Otherwise, a full installation based on Oracle home is assumed.

**Table 2-3 Instant Client Light Shared Libraries Linux and UNIX**

| Linux and UNIX | Description for Linux and UNIX |
|---|---|
| `libclntsh.so.18.1`<br>`libclntshcore.so.18.1` | Client Code Library |
| `libociicus.so` | OCI Instant Client Light Data Shared Library |
| `libnnz18.so` | Security Library |
| `libmql1.so` | Light Weight IPC Library |
| `libipc1.so` | Message Queue Light Library |

**Table 2-4 Instant Client Light Shared Libraries for Microsoft Windows**

| Microsoft Windows | Description for Microsoft Windows |
|---|---|
| `oci.dll` | Forwarding functions that applications link with |
| `oraociicus18.dll` | Data and code |
| `orannzsbb18.dll` | Security Library |
| `oci.sym`<br>`oraociicus18.sym`<br>`orannzsbb18.sym` | Symbol tables |

**See Also:**

Instant Client Shared Libraries for Linux and UNIX Table 2-1

Instant Client Shared Libraries for Microsoft Windows Table 2-2

## 2.6.3 Installing Oracle Instant Client Light

How can Instant Client Light can be installed.

Instant Client Light can be installed in one of these ways:

- Installed from Oracle Technology Network (OTN)

  The following URL is the Instant Client page on the Oracle Technology Network website: Oracle Instant Client

  For Instant Client Light, download and unzip the `basiclite.zip` package in to an empty `instantclient_12_2` directory.

- Installed from an Instant Client Administrator installation

  From the `ORACLE_HOME/instantclient/light` subdirectory, copy `libociicus.so` (or `oraociicus12.dll` on Windows). The Instant Client directory on the `LD_LIBRARY_PATH` (`PATH` on Windows) should contain the Instant Client Light Data Shared Library, `libociicus.so` (`oraociicus12.dll` on Windows), instead of the larger OCI Instant Client Data Shared Library, `libociei.so` (`oraociei12.dll` on Windows).

- Installed from an Oracle Universal Installer installation

  When you select the Instant Client option from the Oracle Universal Installer, `libociei.so` (or `oraociei12.dll` on Windows) is installed in the base directory of the installation, which means these files are placed on the `LD_LIBRARY_PATH` (`PATH` on Windows).

  The Instant Light Client Data Shared Library, `libociicus.so` (or `oraociicus12.dll` on Windows), is installed in the `light` subdirectory of the base directory and not enabled by default. Therefore, to operate in the Instant Client Light mode, the OCI Data Shared Library, `libociei.so` (or `Oraociei12.dll` on Windows) must be deleted or renamed and the Instant Client Light library must be copied from the `light` subdirectory to the base directory of the installation.

  For example, if Oracle Universal Installer has installed the Instant Client in `my_oraic_12_2` directory on the `LD_LIBRARY_PATH` (`PATH` on Windows), then use the following command sequence to operate in the Instant Client Light mode:

  ```
  cd my_oraic_12_2
  rm libociei.so
  mv light/libociicus.so .
  ```

  > **Note:**
  >
  > To ensure that no incompatible binaries exist in the installation, always copy and install the Instant Client files in to an empty directory.

## 2.7 About Patching Oracle Instant Client Shared Libraries on Linux or UNIX

Because Instant Client is a deployment feature, the number and size of files (client footprint) required to run an OCI application has been reduced.

Hence, all files needed to patch Instant Client shared libraries are not available in an Instant Client deployment. A complete client installation based on Oracle home is needed for patching. Use the `opatch` utility for patching.

After you apply the patch in an Oracle home environment, copy the files listed in About Oracle Instant Client to the instant client directory, as described in Install from ZIP Files, from RPMs, and from Oracle Universal Installer.

Instead of copying individual files, you can generate Instant Client zip and RPM files for OCI and OCCI, JDBC, and SQL*Plus as described in Regeneration of Data Shared Library and Zip and RPM Files. Then, you can copy the zip and RPM files to the target system and unzip them as described in Install from ZIP Files, from RPMs, and from Oracle Universal Installer.

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
```

If the Instant Client deployment system does not have the `genezi` utility, you can copy it from the *`ORACLE_HOME`*`/bin` directory.

> **Note:**
>
> The `opatch` utility is not available on Windows.

## 2.8 Regeneration of Data Shared Library and Zip and RPM Files

The process to regenerate the data shared library and the zip and RPM files changed in Oracle Database 12*c* Release 1 (12.1).

Separate make file targets are used to create the data shared libraries, zip, and RPM files either individually or all at once. Currently, `ilibociei` builds only the zip and RPM files. Regeneration of data shared libraries requires both a compiler and linker, which may not be available on all installations. The sections that follow show the make file target used to regenerate data shared libraries, zip, and RPM files.

**Note:**

The regenerated Instant Client binaries contain only the Instant Client files installed in the Oracle Client Administrator Home from which the regeneration is done. Therefore, error messages, character set encodings, and time zone files that are present in the regeneration environment are the only ones that are packaged in the data shared libraries. Error messages, character set encodings, and time zone files depend on which national languages were selected for the installation of the Oracle Client Administrator Home.

Regeneration of the data shared library and the zip and RPM files is not available on Windows platforms.

This section includes the following topics:

- Regenerating Data Shared Library libociei.so
- Regenerating Data Shared Library libociicus.so
- Regenerating Data Shared Libraries libociei.so and libociicus.so in One Step
- Regenerating Zip and RPM Files for the Basic Package
- Regenerating Zip and RPM Files for the Basic Light Package
- Regenerating Zip and RPM Files for the JDBC Package
- Regenerating Zip and RPM Files for the ODBC Package
- Regenerating Zip and RPM Files for the SQL*Plus Package
- Regenerating Zip and RPM Files for the Tools Package
- Regenerating Zip and RPM Files for All Packages

## 2.8.1 Regenerating Data Shared Library libociei.so

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

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk igenlibociei
```

The new regenerated `libociei.so` is placed in the `ORACLE_HOME/instantclient` directory. The original existing `libociei.so` located in this same directory is renamed to `libociei.so0`.

## 2.8.2 Regenerating Data Shared Library libociicus.so

To regenerate Instant Client Light data shared library (`libociicus.so`), use the following commands:

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

The newly regenerated `libociicus.so` is placed in the `ORACLE_HOME/instantclient/light` directory. The original existing `libociicus.so` located in this same directory is renamed to `libociicus.so0`.

## 2.8.3 Regenerating Data Shared Libraries libociei.so and libociicus.so in One Step

To regenerate the data shared libraries `libociei.so` and libociicus.so, use the following commands:

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

The newly regenerated `libociei.so` is placed in the `ORACLE_HOME/instantclient` directory. The original existing `libociei.so` located in this same directory is renamed to `libociei.so0`.

The newly regenerated `libociicus.so` is placed in the `ORACLE_HOME/instantclient/light` directory. The original existing `libociicus.so` located in this same directory is renamed to `libociicus.so0`.

## 2.8.4 Regenerating Zip and RPM Files for the Basic Package

To regenerate the zip and RPM files for the basic package, use the following commands:

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ic_basic_zip
```

## 2.8.5 Regenerating Zip and RPM Files for the Basic Light Package

To regenerate the zip and RPM files for the basic light package, use the following commands:

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ic_basiclite_zip
```

## 2.8.6 Regenerating Zip and RPM Files for the JDBC Package

To regenerate the zip and RPM files for the JDBC package, use the following commands:

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ic_jdbc_zip
```

## 2.8.7 Regenerating Zip and RPM Files for the ODBC Package

To regenerate the zip and RPM files for the ODBC package, use the following commands:

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ic_odbc_zip
```

## 2.8.8 Regenerating Zip and RPM Files for the SQL*Plus Package

To regenerate the zip and RPM files for the SQL*Plus package, use the following commands:

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ic_sqlplus_zip
```

## 2.8.9 Regenerating Zip and RPM Files for the Tools Package

To regenerate the zip and RPM files for the tools package, use the following commands:

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ic_tools_zip
```

## 2.8.10 Regenerating Zip and RPM Files for All Packages

To regenerate the zip and RPM files for all packages, use the following commands:

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ilibociei
```

The new zip and RPM files are generated under the following directory:

```
$ORACLE_HOME/rdbms/install/instantclient
```

Regeneration of the data shared library and the zip and RPM files is not available on Windows platforms.

# 3

# Building and Configuring OCI Applications

This chapter describes features about building and configuring OCI applications on Linux, UNIX, and Windows operating systems.

For other supported operating systems, see the platform specific installation guides for more information.

This chapter includes the following topics:

- Header File and Makefile Locations
- Building an OCI Application on Linux and UNIX
- Building an Application on Windows
- Database Connection Strings
- Client and Server Operating with Different Versions of Time Zone Files
- OCI Client-Side Deployment Parameters Using oraaccess.xml
- About Compatibility and Upgrading
- Fault Diagnosability in OCI

## 3.1 Header File and Makefile Locations

The OCI and OCCI header files that are required for OCI and OCCI client application development on Linux and UNIX operating systems reside in the `$ORACLE_HOME/rdbms/public` directory.

These files are available both with the Oracle Database Server installation, and with the Oracle Database Client Administration and Custom installations. See the SDK for Oracle Instant Client for information about it containing C and C++ header files and a Makefile for developing OCI and OCCI applications while in an Instant Client environment.

All demonstration programs and their related header files reside in the `$ORACLE_HOME/rdbms/demo` directory once they are installed. These demonstration files are installable only from the Examples media. See OCI Demonstration Programs for the names of these programs and their purposes.

Several makefiles are provided in the `demo` directory. Each makefile contains comments with instructions on its use in building OCI executables. Oracle recommends that you use these demonstration makefiles whenever possible to avoid errors in compilation and linking.

The `demo_rdbms.mk` file in the `demo` directory and is an example makefile. See the comments on how to build the demonstration OCI programs. The `demo_rdbms.mk` file includes the `$ORACLE_HOME/rdbms/public` directory. Ensure that your own customized makefiles have the `$ORACLE_HOME/rdbms/public` directory in the `INCLUDE` path.

The `ociucb.mk` file is a makefile in `demo` for building a callback shared library.

# 3.2 Building an OCI Application on Linux and UNIX

How to build an OCI application on Linux and UNIX.

This topic describes the features of OCI that apply to building applications on Linux and UNIX in the following topics:

- Oracle Directory Structure
- Demonstration OCI Programs

> **See Also:**
>
> - *Oracle Database Client Installation Guide for Linux* for operating system requirements for x86–64 Linux platforms, supported Oracle Linux and Red Hat Enterprise distributions for x86-64 platforms, and installation requirements for programming environments for Linux x86-64
> - *Oracle Database Instant Client Installation Guide for Apple Mac OS X (Intel)* for checking the software requirements

## 3.2.1 Oracle Directory Structure

The `$ORACLE_HOME` directory contains the following directories described in the following table that are relevant to OCI.

These directories are for the full client and Oracle Database, but not for the Oracle Instant Client. These files include the library files needed to link and run OCI applications, and link with other Oracle products.

**Table 3-1 ORACLE_HOME Directories and Contents**

| Directory Name | Contents |
|---|---|
| `/admin` | Configuration files |
| `/demo` | Sample programs, make files, SQL files, and so forth |
| `/imclude` | Header files |
| `/lib` | Library files |
| `/mesg` | Message files |
| `/public` | Public header files |

## 3.2.2 Demonstration OCI Programs

A set of OCI demonstration programs and their corresponding project files are optionally installed after an Oracle Database installation and set up in the *`ORACLE_BASE/ORACLE_HOME/`*`demo` subdirectory.

Build and run these OCI demonstration programs to familiarize yourself with the steps involved in developing OCI applications.

To build a demo OCI program, run the `make` file (`demo_rdbms.mk`) located in the `/demo` directory. For example, to build a single OCI demo, use the following `make` command syntax:

```
make -f demo_rdbms.mk build EXE=demo OBJS="demo.o ..."
```

For example, to build the OCI `cdemo81.c` program, enter the following `make` command:

```
make -f demo_rdbms.mk build EXE=cdemo81 OBJS=cdemo81.o
```

In this example, the executable file is created or updated from the object file, which in turn is made by compiling the source file `cdemo81.c`.

**See Also:**

- *Oracle Database Examples Installation Guide* for information about installing the demonstration OCI programs using Oracle Universal Installer
- Downloads: Oracle Database 12*c* Release 2 (12.2.0.1.0) Standard Edition 2 and Enterprise Edition for a link to the Oracle Database 12*c* Release 2 Examples (12.2.0.1.0) for Linux x86-64 `linuxx64_12201_examples.zip`
- Review the contents of the `demo_rdbms.mk` file to learn more about running the many other OCI demonstration programs that are available in the `demo` subdirectory.
- OCI Demonstration Programs for more information about OCI demonstration programs

## 3.3 Building an Application on Windows

How to build an OCI application on Windows.

See Getting Started with OCI for Windows for complete information.

**See Also:**

*Oracle Database Client Installation Guide for Microsoft Windows* for information about Oracle Database software client requirements.

## 3.4 Database Connection Strings

This topic describes Oracle Net naming methods that do not use `ORACLE_BASE_HOME`, or `ORACLE_HOME` (to locate configuration files such as `tnsnames.ora`, `sqlnet.ora`, or `oraaccess.xml`).

In particular, the `connect_identifier` in the `OCIServerAttach()` call can be specified in the following formats:

- An easy connect naming string is of the form:

  ```
  [[protocol:]//]host1{,host12}[:port1]{,host2:port2}[/service_name]
  [:server][/instance_name][?parameter_name=value{&parameter_name=value}]
  ```

  Starting with Oracle Database Release 19c, the easy connect naming syntax has been extended, see Easy Connect Naming Enhancements.
- As a local naming parameters `tnsnames.ora` configuration file entry of the form:

  ```
  net_service_name=
   (DESCRIPTION=
  (ADDRESS=(protocol_address_information))
     (CONNECT_DATA=
       (SERVICE_NAME=service_name)))
  ```
- As an Oracle Net connect descriptor of the form:

  ```
  "(DESCRIPTION=(ADDRESS=(PROTOCOL=protocol-name) (HOST=host-name) (PORT=port-
  number))
  (CONNECT_DATA=(SERVICE_NAME=service-name)))"
  ```
- A Connection Name that is resolved through Directory Naming where the site is configured for LDAP server discovery.

For naming methods such as `tnsnames.ora` and directory naming to work, the `TNS_ADMIN` environment variable can also be set.

If the `TNS_ADMIN` environment variable is not set, and `tnsnames.ora` 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`. So it is preferred and easier to set the `TNS_ADMIN` environment variable to establish the location of the `tnsnames.ora` file.

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 a Windows operating system, the `LOCAL` environment variable is used instead of `TWO_TASK`.

Similarly, for OCI 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.

**See Also:**

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

## 3.4.1 Examples of Oracle Database Connection String Connect Identifiers

If you are using OCI applications, for example SQL*Plus, then you can specify the database connection string in the following ways:

If the `listener.ora` file on the Oracle database 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 OCI application, connect identifier is:

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

The connect identifier can also be specified as:

```
"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=1573))(CONNECT_DATA=
(SERVICE_NAME=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:

```
export TWO_TASK=//server6:1573/rdbms3.server6.us.alchemy.com
```

You can also specify the `TWO_TASK` environment variable as:

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

Then you can invoke the OCI application with an empty connect identifier. For example, to run SQL*Plus:

```
sqlplus user
```

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)))
```

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

```
export TNS_ADMIN=/home/webuser/myconfigs
```

Then you can use the connect identifier `conn_str` for invoking OCI application, for example 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 preceding `tnsnames.ora` file is located in an installation based Oracle home, in the `/network/server6/home/dba/oracle/network/admin` directory, then the `ORACLE_HOME` environment variable can be set as follows and OCI application, for example SQL*Plus can be invoked as previously, with the identifier `conn_str`:

```
export ORACLE_HOME=/network/server6/home/dba/oracle
```

Finally, if `tnsnames.ora` can be located by `TNS_ADMIN` or `ORACLE_HOME`, then the `TWO_TASK` environment variable can be set as follows enabling you to invoke OCI application, for example SQL*Plus without a connect identifier:

```
export TWO_TASK=conn_str
```

## 3.5 Client and Server Operating with Different Versions of Time Zone Files

In Oracle Database Release 11.2 (or later) you can use different versions of the time zone file on the client and server; this mode of operation was not supported before Oracle Database Release 11.2.

Both client and server must be 11.2 or later to operate in such a mixed mode. This section discusses the ramifications of operating in such a mode. To avoid these ramifications use the same time zone file version for client and server.

The following behavior is seen when the client and server use different time zones file versions. Note that the use of different time zone file versions only affects the handling of `TIMESTAMP WITH TIMEZONE` (`TSTZ`) data type values.

- The OCI Datetime and Interval APIs listed here unconditionally raise an error when the input parameters are of `TSTZ` type. This is because these operations depend on the local time zone file on the client that is not synchronized with the database. Continuing with the computation in such a configuration can result in inconsistent computations across the client and database tiers.

  OCIDateTimeCompare()
  OCIDateTimeConstruct()
  OCIDateTimeConvert()
  OCIDateTimeSubtract()
  OCIIntervalAdd()
  OCIIntervalSubtract()
  OCIIntervalFromTZ()
  OCIDateTimeGetTimeZoneName()
  OCIDateTimeGetTimeZoneOffset()[1]
  OCIDateTimeSysTimeStamp()

- There is a performance penalty when you retrieve or modify `TSTZ` values. The performance penalty arises because of the additional conversions needed to compensate for the client and server using different time zone file versions.
- If new time zone regions are defined by the more recent time zone file, you can see an error operating on a `TIMESTAMP WITH TIMEZONE` value belonging to the new region on a node that has a time zone file version that does not recognize the new time zone region.

Applications that manipulate opaque type or `XMLType` instances or both containing `TSTZ` type attributes must use the same time zone file version on client and server to avoid data loss.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information about upgrading the time zone file and timestamp with time zone data

## 3.6 OCI Client-Side Deployment Parameters Using oraaccess.xml

This topic describes the OCI client-side deployment parameters using oraaccess.xml.

This section includes the following topics:

- About oraaccess.xml
- About Client-Side Deployment Parameters Specified in oraaccess.xml
- High Level Structure of oraaccess.xml
- About Specifying Global Parameters in oraaccess.xml
- About Specifying Defaults for Connection Parameters
- Overriding Connection Parameters at the Connection-String Level
- About OCI Session Pool Configuration in oraaccess.xml
- File (oraaccess.xml) Properties

### 3.6.1 About oraaccess.xml

Starting with Oracle Database Release 12*c* Release 1 (12.1), Oracle provides an `oraaccess.xml` file, a client-side configuration file.

You can use the `oraaccess.xml` file to configure selected OCI parameters (some of which are accepted programatically in various OCI API calls), thereby allowing OCI behavior to be changed during deployment without modifying the source code that calls OCI.

---

[1] Returns an `ORA-01805` error when timezone files on the client and server do not match (regions are not synchronized); returns `OCI_SUCCESS` when region time zone values are the same (represent the same instant in UTC), though the `TIME ZONE` offsets are different.

Updates to the `oraaccess.xml` file will not affect already running clients. In order to pick up any updates to the `oraaccess.xml` file, already running clients need to be restarted.

The `oraaccess.xml` file is read from the directory specified by the `TNS_ADMIN` environment variable in regular and instant client installations. This is the *`$ORACLE_HOME`*`/network/admin` directory on UNIX operating systems and the *`%ORACLE_HOME%`*`\NETWORK\ADMIN` directory on Microsoft Windows operating systems, if `TNS_ADMIN` is not set in regular client installations.

## 3.6.2 About Client-Side Deployment Parameters Specified in oraaccess.xml

When equivalent parameters are set both in the `sqlnet.ora` and `oraaccess.xml` files, the `oraaccess.xml` file setting takes precedence over the corresponding `sqlnet.ora` file setting.

In such cases, Oracle recommends using the `oraaccess.xml` file settings moving forward. For any network configuration, the `sqlnet.ora` file continues to be the primary file as network level settings are not supported in the `oraaccess.xml` file.

## 3.6.3 High Level Structure of oraaccess.xml

Describes the high-level structure of the `oraaccess.xml` file.

The `oraaccess.xml` file has a top-level node `<oraaccess>` with the following three elements:

- `<default_parameters>` - This element describes any default parameter settings shared across connections. These default parameters include:
  - Defaults for global parameters - These global parameters can only be specified once and hence are applicable to all connections and cannot be overridden at the connection level. These parameters are specified using the following tags:
    - `<events>` - Creates the OCI Environment in `OCI_EVENTS` mode, which is required for Fast Application Notification (FAN) and runtime connection load balancing
    - `<result_cache>` - Sets OCI client result cache parameters
    - `<diag>` - Sets OCI fault diagnosability parameters
  - Defaults for connection-specific parameters - Connection parameters can be set to different values for specific connections. However, they too can be defaulted, and overridden on a per connection string basis as needed. These defaults are shared across all connections (unless overridden at the connection level, which is described in the `<config_descriptions>` list item) that follows. These defaults are specified by the following tags:
    - `<prefetch>` - Sets the number of prefetch rows for all queries; specified using the `<rows>` parameter.
    - `<statement_cache>` - Sets the maximum number of statements that can be cached per session; specified using the `<size>` parameter.

  - `<auto tune>` - Consists of: `<enable>` to turn auto tuning on or off; `<ram_threshold>`, which sets the memory threshold for auto-tuning to stop using more memory when available physical memory on the client system has reached this threshold; and `<memory_target>`, which sets the memory limit that OCI auto-tuning can use per client process.
  - `<fan_subscription_failure_action>` - Sets the action upon subscription failure to be either the value `trace` or `error`.
  - `<ons>` - Sets a variety of ONS client-side deployment configuration parameters used for FAN notifications.
- `<config_descriptions>` - This element associates a configuration alias element (`<config_alias>`), which is basically a name, with a specific set of parameters (`<parameters>`) that contain one or more connection parameters. These connection parameters are the same connection parameters within the element `<default_parameters>` described previously, namely: `<prefetch>`, `<statement_cache>`, `<auto_tune>`, `<fan_subscription_failure_action>` and `<ons>`.
- `<connection_configs>` - This element associates one or more connection strings used by an application with a config alias, thus allowing multiple connection string elements to share the same set of parameters.

  A connection configuration element (`<connection_config>`) associates a connection string element (`<connection_string>`) with a configuration alias element (`<config_alias>`).

  A connection string is indirectly associated with a set of parameters through the configuration alias, which allows multiple connection string elements to share the same set of parameters.

The sections that follow describe these client-side deployment parameters in more detail.

> **See Also:**
>
> About Specifying Defaults for Connection Parameters

## 3.6.4 About Specifying Global Parameters in oraaccess.xml

As described, the `<default_parameters>` tag allows specifying default values for various OCI parameters.

Of these, some parameters can only be specified once and hence apply to all connections. These global parameters are described using the following tags:

- `<events>`

  This creates the OCI Environment in `OCI_EVENTS` mode, which is required for Fast Application Notification (FAN) and runtime connection load balancing.

  ```
  <events>
    true <!--value could be false also -->
  </events>
  ```
- `<result_cache>`

  - `<max_rset_rows>` - Maximum size of any result set in rows in the per-process query cache. Equivalent to `OCI_RESULT_CACHE_MAX_RSET_ROWS` in the `sqlnet.ora` file.
  - `<max_rset_size>` - Maximum client result cache size. Set the size to 32,768 bytes (32 Kilobytes (KB)) or greater. Equivalent to `OCI_RESULT_CACHE_MAX_RSET_SIZE` in the `sqlnet.ora` file.
  - `<max_size>` - Maximum size in bytes for the per-process query cache. Specifying a size less than 32,768 in the client disables the client result cache feature. Equivalent to `OCI_RESULT_CACHE_MAX_SIZE` in the `sqlnet.ora` file.

```
<result_cache>
      <max_rset_rows>10</max_rset_rows>
      <max_rset_size>65535</max_rset_size>
      <max_size>65535</max_size>
</result_cache>
```

When equivalent parameters are set both in the `sqlnet.ora` and `oraaccess.xml` files, the `oraaccess.xml` file setting takes precedence over the corresponding `sqlnet.ora` file setting.

See Table 3-2 for a listing of equivalent OCI parameter settings.

- `<diag>`

You can specify the following elements:

  - `<adr_enabled>` - Enables or disables diagnosability. Equivalent to `DIAG_ADR_ENABLED` in the `sqlnet.ora` file. Values: true or false.
  - `<dde_enabled>` - Enables or disables DDE. Values: true or false.
  - `<adr_base>` - Sets the ADR base directory, which is a system-dependent directory path string to designate the location of the ADR base to use in the current ADRCI session. Equivalent to `ADR_BASE` in the `sqlnet.ora` file. Value: directory path for ADR base directory.
  - `<sighandler_enabled>` - Enables or disables OCI signal handler. Values: true or false.
  - `<restricted>` - Enables or disables full dump files. Oracle Database client contains advanced features for diagnosing issues, including the ability to dump diagnostic information when important errors are detected. By default, these dumps are restricted to a small subset of available information, to ensure that application data is not dumped. However, in many installations, secure locations for dump files may be configured, ensuring the privacy of such logs. In such cases, it is recommended to turn on full dumps; this can greatly speed resolution of issues. Full dumps can be enabled by specifying a value of false. Values: true or false.
  - `<trace_events>` - Indicates the trace event number and the level of tracing to be in effect. Currently only event 10883 is supported. The available levels are 5 and 10.

```
<diag>
   <adr_enabled>false</adr_enabled>
   <dde_enabled>false</dde_enabled>
   <adr_base>/foo/adr</adr_base>
   <sighandler_enabled>false</sighandler_enabled>
   <restricted>true</restricted>
   <trace_events>
```

```
      <trace_event>
        <number>10883</number>
        <level>5</level>
      </trace_event>
    </trace_events>
</diag>
```

When equivalent parameters are set both in the `sqlnet.ora` and `oraaccess.xml` files, the `oraaccess.xml` file setting takes precedence over the corresponding `sqlnet.ora` file setting.

See Table 3-2 for a listing of equivalent OCI parameter settings.

**Table 3-2 Equivalent OCI Parameter Settings in oraaccess.xml and sqlnet.ora**

| Parameter Group | oraaccess.xml Parameters | sqlnet.ora Parameters |
|---|---|---|
| OCI client result cache | `<max_rset_rows>` | `OCI_RESULT_CACHE_MAX_RSET_ROWS` |
| OCI client result cache | `<max_rset_size>` | `OCI_RESULT_CACHE_MAX_RSET_SIZE` |
| OCI client result cache | `<max_size>` | `OCI_RESULT_CACHE_MAX_SIZE` |
| OCI fault diagnosability | `<adr_enabled>` | `DIAG_ADR_ENABLED` |
| OCI fault diagnosability | `<dde_enabled>` | `DIAG_DDE_ENABLED` |
| OCI fault diagnosability | `<adr_base>` | `ADR_BASE` |

**See Also:**

- *Oracle Database Development Guide* for information about deployment time settings for client result cache and client configuration file parameters
- *Oracle Database Net Services Reference* for more information about ADR diagnostic parameters in the `sqlnet.ora` file

## 3.6.5 About Specifying Defaults for Connection Parameters

Describes the default values you can set for connection parameters shared across connections.

You can specify the following connection parameters that are shared across connections:

- `<prefetch>` - Specifies prefetch row count for `SELECT` statements.

```
<prefetch>
  <rows>100</rows>
</prefetch>
```

Setting this parameter appropriately can help reduce round-trips to the database, thereby improving application performance.

Note that this only overrides the `OCI_ATTR_PREFETCH_ROWS` parameter (whether explicitly specified by the application or not). If the application has specified `OCI_ATTR_PREFETCH_MEMORY` explicitly, then the actual prefetch row count will be determined by using both constraints. The `OCI_ATTR_PREFETCH_MEMORY` constraint equivalent cannot be specified in the `oraaccess.xml` file.

Also note that OCI prefetching may still get disabled if the `SELECT` statement fetches columns of specific data types. For more details, see About Fetching Results for information about limitations of OCI prefetch.

- `<statement_cache>` - Specifies the number of OCI Statement handles that can be cached per session.

  ```
  <statement_cache>
    <size>100</size>
  </statement_cache>
  ```

  Caching statement handles improves repeat execute performance by reducing client side and server side CPU consumption and network traffic.

  Note that for this parameter to take effect, the application must be programmed to use `OCIStatementPrepare2()` and `OCIStatementRelease()` calls (and not the older `OCISatementPrepare()` and `OCIHandleFree()` equivalents for getting and disposing of statement handles.

- `<auto_tune>` - Used to enable OCI Auto tuning.

  ```
  <auto_tune>
    <enable>true</enable>
    <ram_threshold>0.1</ram_threshold><!--percentage -->
    <memory target>2M</memory_target>
  </auto_tune>
  ```

  Enabling auto-tuning can help OCI automatically tune the statement-cache size based on specified memory constraints. This can help dynamically tune the statement cache size to an appropriate value based on runtime application characteristics and available memory resources.

  Note that for auto tuning OCI Statement Cache, the application must be programmed to use `OCIStatementPrepare2()` and `OCIStatementRelease()` calls (and not the older `OCISatementPrepare()` and `OCIHandleFree()` equivalents for getting and disposing of statement handles.

- `<fan_subscription_failure_action>` - Used to determine how OCI responds to a failure to subscribe for FAN notifications.

  A value of `trace` records any failure to subscribe for FAN notifications (if FAN is enabled) in the trace file and OCI proceeds ignoring the failure. A value of `error` makes OCI return an error if an attempt to subscribe for FAN notifications fails.

  ```
  <fan>
    <!--only possible values are "trace" and "error" -->
    <subscription_failure_action>
      trace
  ```

```
  </subscription_failure_action>
</fan>
```

- `<ons>` - Sets up Oracle Notification Service (ONS) parameters.

  You can specify the following connection parameters:

  - `<subscription_wait_timeout>` - Length of time in seconds the client waits for its subscription to the ONS server.
  - `<auto_config>` - true or false. If true, the configuration specified in this section will augment the auto configuration information that the client receives from the database. If false, it will override the same.
  - `<thread_stack_size>` - Size in bytes of the event notification thread stack.
  - `<debug>` - true or false. Whether debug mode is on (true) or off (false).
  - `<wallet_location>` - Directory that contains an auto logon wallet file for a secure ONS subscription.
  - `<servers>` - Host list with ports and connection distribution.

```
<ons>
  <!--values or in seconds -->
  <subscription_wait_timeout>
    5
  </subscription_wait_timeout>
  <auto_config>true</auto_config> <!--boolean -->
  <threadstacksize>100k</threadstacksize>
  <debug>true</debug>
  <wallet_location>/etc/oracle/wallets/</wallet_location>
  <servers>
    <address_list>
       <name>pacific</name>
       <max_connections> 3 <\max_connections>
       <hosts>
         10.228.215.121:25293,
         10.228.215.122:25293
       </hosts>
    </address_list>
    <address_list>
       <name>Europe</name>
       <max_connections>3<\max_connections>
       <hosts>
         myhost1.mydomain.com:25273,
         myhost2.mydomain.com:25298,
         myhost3.mydomain.com:30004
       </hosts>
    </address_list>
  </servers>
</ons>
```

**See Also:**

- <auto_tune>
- <ram_threshold>
- <memory_target>
- *Oracle Universal Connection Pool Developer’s Guide* for information about ONS configuration parameters

## 3.6.6 Overriding Connection Parameters at the Connection-String Level

Using the `oraaccess.xml` file also allows you to override the very same set of connection-specific parameters at the connection-string level as well.

This allows for overriding those connection-specific parameters based on requirements of individual applications.

Using the `<config_descriptions>` tag, you can specify a set of connection-specific parameters (`<parameters>`) to be associated with a configuration alias (`<config_alias>`, which creates a named group of connection-specific parameters). Thereafter, using the `<connection_configs>` tag, you can associate one or more connection-strings (specified using the `<connection-string>` tag) with a `<config_alias>`. This permits a level of indirection that allows multiple `<connection_string>` elements to share the same set of `<parameters>`.

**Example 1**

This example shows a very simple `oraaccess.xml` file configuration that highlights defaulting of global and connection parameters applicable across all connections.

```
<?xml version="1.0" encoding="ASCII" ?>
<!--
     Here is a sample oraaccess.xml.
     This shows defaulting of global and connection parameters
     across all connections.
-->
 <oraaccess xmlns="http://xmlns.oracle.com/oci/oraaccess"
  xmlns:oci="http://xmlns.oracle.com/oci/oraaccess"
  schemaLocation="http://xmlns.oracle.com/oci/oraaccess
  http://xmlns.oracle.com/oci/oraaccess.xsd">
  <default_parameters>
    <prefetch>
      <rows>50</rows>
    </prefetch>
    <statement_cache>
      <size>100</size>
    </statement_cache>
    <result_cache>
      <max_rset_rows>100</max_rset_rows>
      <max_rset_size>10K</max_rset_size>
      <max_size>64M</max_size>
    </result_cache>
  </default_parameters>
</oraaccess>
```

### Example 2

This example shows connection parameters being overridden at the connection level.

```
<?xml version="1.0" encoding="ASCII" ?>
<!--
        Here is a sample oraaccess.xml.
        This highlights some connection parameters being
        overridden at the connection level
     -->
 <oraaccess xmlns="http://xmlns.oracle.com/oci/oraaccess"
  xmlns:oci="http://xmlns.oracle.com/oci/oraaccess"
  schemaLocation="http://xmlns.oracle.com/oci/oraaccess
  http://xmlns.oracle.com/oci/oraaccess.xsd">
  <default_parameters>
    <prefetch>
      <rows>50</rows>
    </prefetch>
    <statement_cache>
      <size>100</size>
    </statement_cache>
    <auto_tune>
      <enable>true</enable>
      <ram_threshold>2.67</ram_threshold>
      <memory_target>204800</memory_target>
    </auto_tune>
    <result_cache>
      <max_rset_rows>100</max_rset_rows>
      <max_rset_size>10K</max_rset_size>
      <max_size>64M</max_size>
    </result_cache>
  </default_parameters>
    <!--
         Create configuration descriptions, which are
         groups of connection parameters associated with
         a config_alias.
    -->
  <config_descriptions>
    <config_description>
      <config_alias>bar</config_alias>
      <parameters>
        <prefetch>
          <rows>20</rows>
        </prefetch>
      </parameters>
    </config_description>
    <config_description>
      <config_alias>foo</config_alias>
      <parameters>
        <statement_cache>
          <size>15</size>
        </statement_cache>
      </parameters>
    </config_description>
  </config_descriptions>
  <!--
         Now map the connection string used by the application
         with a config_alias.
  -->
  <connection_configs>
    <connection_config>
      <connection_string>hr_db</connection_string>
      <config_alias>foo</config_alias>
    </connection_config>
    <connection_config>
      <connection_string>finance_db</connection_string>
      <config_alias>bar</config_alias>
    </connection_config>
  </connection_configs>
</oraaccess>
```

### Example 3

This example highlights setup for FAN notifications.

```
<?xml version="1.0" encoding="ASCII" ?>
   <!--
    Here is a sample for oraaccess.xml for
    setting up for FAN notifications.
   -->
  <oraaccess xmlns="http://xmlns.oracle.com/oci/oraaccess"
    xmlns:oci="http://xmlns.oracle.com/oci/oraaccess"
    schemaLocation="http://xmlns.oracle.com/oci/oraaccess
    http://xmlns.oracle.com/oci/oraaccess.xsd">
    <default_parameters>
      <fan>
        <!-- only possible values are "trace" or "error" -->
        <subscription_failure_action>
          error
        </subscription_failure_action>
      </fan>
      <ons>
        <subscription_wait_timeout>
          5
        </subscription_wait_timeout>
        <auto_config>true</auto_config>
      </ons>
    <events>true</events>
    </default_parameters>
  </oraaccess>
```

### Example 4

This example highlights an advanced `oraaccess.xml` file configuration usage with manual ONS settings. Manual ONS settings should be used rarely.

```
  <?xml version="1.0" encoding="ASCII" ?>
    <!--
       Here is a sample for oraaccess.xml that highlights
       manual ONS settings.
    -->
  <oraaccess xmlns="http://xmlns.oracle.com/oci/oraaccess"
    xmlns:oci="http://xmlns.oracle.com/oci/oraaccess"
    schemaLocation="http://xmlns.oracle.com/oci/oraaccess
    http://xmlns.oracle.com/oci/oraaccess.xsd">
    <default_parameters>
      <fan>
       <!-- only possible values are "trace" or "error"  -->
        <subscription_failure_action>
        error
      </subscription_failure_action>
    </fan>
    <ons>
      <subscription_wait_timeout>
        5
      </subscription_wait_timeout>
      <auto_config>true</auto_config>
      <!--This provides the manual configuration for ONS.
          Note that this functionality should not need to be used
          as auto_config can normally discover this
          information. -->
      <servers>
        <address_list>
          <name>pacific</name>
          <max_connections>3</max_connections>
          <hosts>10.228.215.121:25293, 10.228.215.122:25293</hosts>
        </address_list>
        <address_list>
          <name>Europe</name>
          <max_connections>3</max_connections>
          <hosts>myhost1.mydomain.com:25273,
            myhost2.mydomain.com:25298,
            myhost3.mydomain.com:30004</hosts>
        </address_list>
      </servers>
    </ons>
    <events>true</events>
  </default_parameters>
</oraaccess>
```

> **See Also:**
>
> About Specifying Defaults for Connection Parameters about overriding the very same set of connection-specific parameters at the connection-string level

## 3.6.7 About OCI Session Pool Configuration in oraaccess.xml

Beginning with Oracle Database release 18c, version 18.1, the OCI session pool configuration can be set up using the oraaccess.xml client-side configuration file.

The OCI session pool is configured by specifying the following set of parameters in the `oraaccess.xml` configuration file. These parameters can be specified in the default parameters section or in the configuration descriptions section. If specified in the default parameters section, then it applies to all the session pools in the application. These settings can benefit any OCI Session Pool allowing OCI Session Pool settings to be overridden.

- `<session_pool>` — Sets up session pool parameters.
  - `<enable>` — Setting this to true makes the session pool configuration effective. This is a mandatory parameter, that means, if the `<session_pool>` parameter is configured, then `<enable>` parameter must also be configured.
- `<min_size>` — Minimum number of connections in the pool. The default is 0.

- `<max_size>` — Maximum number of connections in the pool. This is a mandatory parameter, that means, if the `<session_pool>` parameter is configured, then `<max_size>` parameter must also be configured.
- `<increment>` — Amount of increase in the number of connections in the pool as the pool expands. The default is 1.
- `<inactivity_timeout>` — Maximum time in seconds for which a connection stays idle in the pool, after which it is terminated. The default is 0, that means, there is no limit for which a connection stays idle in the pool.
- `<max_use_session>` — Maximum number of times a connection can be taken and released to the pool. The default is 0, that means, there is no limit to take a connection and release to the pool.
- `<max_life_time_session>` — Time, in seconds, a connection will stay after it has been created in the pool. The default is 0, that means, there is no limit for a connection to stay after it has been created in the pool.

Using the `oraaccess.xml` file allows you to configure an OCI session pool for each needed connection service. The following example shows two groups of connection parameters associated with its respective config alias, the `sales_config` and the `hr_config`, where each connection string that the application uses is mapped with its respective config alias, thus providing two OCI session pools.

```
<oraaccess xmlns="http://xmlns.oracle.com/oci/oraaccess"
            xmlns:oci="http://xmlns.oracle.com/oci/oraaccess"
            schemaLocation="http://xmlns.oracle.com/oci/oraaccess
            http://xmlns.oracle.com/oci/oraaccess.xsd">
  <default_parameters>
  </default_parameters>
    <!--
         Create configuration descriptions, which are
         groups of connection parameters associated with
         a config_alias.
    -->
  <config_descriptions>
    <config_description>
      <config_alias> sales_config </config_alias>
      <parameters>
         <session_pool>
            <enable>true</enable>
            <min_size> 10 </min_size>
            <max_size> 100 </max_size>
            <increment> 5 </increment>
         </session_pool>
      </parameters>
    </config_description>
    <config_description>
      <config_alias> hr_config </config_alias>
      <parameters>
         <session_pool>
            <enable>true</enable>
            <max_size> 10 </max_size>
         </session_pool>
      </parameters>
    </config_description>
  </config_descriptions>
  <!--
         Now map the connection string used by the application
         with a config_alias.
  -->
  <connection_configs>
    <connection_config>
      <connection_string>sales.us.acme.com</connection_string>
      <config_alias>sales_config</config_alias>
    </connection_config>
    <connection_config>
      <connection_string>hr.us.acme.com</connection_string>
      <config_alias>hr_config</config_alias>
    </connection_config>
  </connection_configs>
</oraaccess>
```

## 3.6.8 File (oraaccess.xml) Properties

Lists some high level rules with regards to the `oraaccess.xml` file syntax stated here for simplicity.

The XML schema specified in the `oraaccess.xsd` file is the ultimate formal reference for oraaccess syntax:

- The contents of the file are case sensitive, and all elements (tags, parameter names) are in lower case.
- Comments are allowed between parameters (nodes); for example, Comment "<!-- comments -->".
- For the syntax with respect to the order of the parameters, see the XML Schema: `oraaccess.xsd` file (see information about the `oraaccess.xsd` file later in this list).
- For memory size, valid values and formats are 100, 100k, 100K, 1000M, and 1121m. This means only suffixes 'M', 'm', 'K', 'k', or no suffix are allowed. 'K' or 'k' means kilobytes and 'M' or 'm' means megabytes. No suffix means the size is in bytes.
- <ram_threshold> should be a decimal number between 0 and 100 and indicates a percentage value.
- Where a number is expected, only positive unsigned integers are allowed; no sign is allowed. An example of a valid number usage is `<statement_cache> <size>100</size> </statement_cache>`).
- Configuration alias names (`<config_alias>foo</config_alias>`) are not case-sensitive
- String parameters (such as `<config_alias>`) are not expected to be quoted.
- These rules are all encapsulated in the schema definition.
- OCI will report an error if OCI is provided an invalid `oraaccess.xml` file.
- Before deploying an `oraaccess.xml` file, Oracle recommends that you validate it with the Oracle supplied XML schema file: `oraaccess.xsd`. The schema file is installed under `ORACLE_HOME/rdbms/admin` in a regular client and under `instantclient_12_2/sdk/admin` in an instant client SDK. Customers can use their

own favorite XML validation tools to perform the validation after modifying the `oraaccess.xml` file.

- Sample `oraaccess.xml` files can be found in the `ORACLE_HOME/rdbms/demo` directory in a regular client and in the `instantclient_12_2/sdk/demo` in an instant client. The parameters in these files are for demonstration purpose only and should be modified and tested as per the application's requirement before deploying it.

# 3.7 About Compatibility and Upgrading

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

- Oracle Client and Server Cross Version Compatibility
- Version Compatibility of Statically Linked and Dynamically Linked Applications
- About Upgrading of Existing OCI Release 7 Applications
- Obsolete OCI Routines
- OCI Routines Not Supported

## 3.7.1 Oracle Client and Server Cross Version Compatibility

As a general guide, Oracle client and server versions are cross version compatible with a number versions.

This means that you can connect to some older and newer versions of the database and do not always have to upgrade both the client and server at the same time.

However, certain products or utilities may impose additional restrictions on supported combinations specific to the product or utility.

**See Also:**

My Oracle Support Document 207303.1 for information about client and server interoperability support matrix for different Oracle versions.

## 3.7.2 Version Compatibility of Statically Linked and Dynamically Linked Applications

Here are the rules for relinking for a new release.

- Statically linked OCI applications:

  Statically linked OCI applications must be relinked for both major and minor releases, because the statically linked Oracle Database 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 is no longer compatible with the statically linked code.
- Dynamically linked OCI applications:

Dynamically linked OCI applications from Oracle Database 10*g* and later releases need not be relinked. That is, the Oracle Database client-side dynamic library is upwardly compatible with the previous version of the library. Oracle Universal Installer creates a symbolic link for the previous version of the library that resolves to the current version. Therefore, an application that is dynamically linked with the previous version of the Oracle Database client-side dynamic library does not need to be relinked to operate with the current version of the Oracle Database client-side library.

**Note:**

If the application is linked with a runtime library search path (such as `-rpath` on Linux), then the application may still run with the version of Oracle Database client-side library it is linked with. To run with the current version of Oracle Database client-side library, it must be relinked.

**See Also:**

- *Oracle Database Upgrade Guide* for information about compatibility and upgrading
- The server versions supported currently are found on My Oracle Support Document 207303.1

### 3.7.3 About Upgrading of Existing OCI Release 7 Applications

OCI has been significantly improved with many features since OCI release 7.

Applications that use the OCI release 7.3 API work unchanged against the current release of Oracle Database. They do need to be linked with the current client library. However, OCI release 7.3 API has been deprecated and this option will not be available in future Oracle releases.

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 relinked Oracle Database release 7 OCI applications work unchanged against a current Oracle Database. This option is deprecated and will not be available in future Oracle releases.
- Upgrade to the current OCI client and modify the application: To use the performance and scalability benefits provided by the current OCI, however, you must modify your existing applications to use the current OCI programming paradigm, rebuild and relink them with the current OCI library, and run them against the current release of the Oracle database.

If you want to use any of the object capabilities of the current Oracle Database release, you must upgrade your client to the current release of OCI.

**Note:**

Applications using version 7 API will not be able to connect to Oracle Database 12*c* by default. Such applications willing to connect to Oracle Database 12*c* must set `sqlnet.allowed_logon_version` to `8`.

In Oracle Database 12*c* Release 2 (12.2), identifier length supported by Oracle Database has been increased from 30 bytes to 128 bytes. With this database change, OCI applications using deprecated V7 API `odessp()` may need to be changed. In `odessp()` for the argument `argnam`, an application needs to pass the two dimensional array, which can accommodate identifiers of length 128 bytes, which was previously 30 bytes.

Applications using UPI API `upidpr()` may also need to be changed with long identifiers in Oracle Database 12*c* Release 2 (12.2). In `upidpr()` for the argument `argument_name`, an application needs to pass the two dimensional array, which can accommodate identifiers of length 128 bytes, which was previously 30 bytes.

**See Also:**

Programmer's Guide to the Oracle7 Server Call Interface in Oracle 7 Documentation.

## 3.7.4 Obsolete OCI Routines

Release 8.0 of the OCI introduced an entirely new set of functions that were not available in release 7.3.

Oracle strongly recommends that new applications use the new calls to improve performance and provide increased functionality. Future releases of Oracle will not be supporting the release 7.3 API.

Table 3-3 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 *Programmer's Guide to the Oracle Call Interface, Release 7.3*.

**Note:**

In many cases the new or current OCI routines do not map directly onto the 7.*x* routines, so it almost 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 or current call is made. See the remaining chapters, in particular OCI Programming Basics of this guide for more information.

**Table 3-3    Obsolescent OCI Functions**

| 7.*x* OCI Routine | Equivalent or Similar Later OCI Routine |
|---|---|
| `obindps()`, `obndra()`, `obndrn()`, `obndrv()` | `OCIBindByName()` or `OCIBindByName2()`, `OCIBindByPos()` or `OCIBindByPos2()` (Note: additional bind calls may be necessary for some data types) |
| `obreak()` | `OCIBreak()` |
| `ocan()` | none |
| `oclose()` | Note: cursors are not used in release 8.x or later |
| `ocof()`, `ocon()` | `OCIStmtExecute()` with `OCI_COMMIT_ON_SUCCESS` mode |
| `ocom()` | `OCITransCommit()` |
| `odefin()`, `odefinps()` | `OCIDefineByPos()` or `OCIDefineByPos2()` (Note: additional define calls may be necessary for some data types) |
| `odescr()` | Note: schema objects are described with OCIDescribeAny(). A describe, as used in release 7.*x*, most often be done by calling OCIAttrGet() on the statement handle after SQL statement execution. |
| `odessp()` | `OCIDescribeAny()` |
| `oerhms()` | `OCIErrorGet()` |
| `oexec()`, `oexn()` | `OCIStmtExecute()` |
| `oexfet()` | `OCIStmtExecute()`, `OCIStmtFetch2()` (Note: result set rows can be implicitly prefetched) |
| `ofen()`, `ofetch()` | `OCIStmtFetch2()` |
| `oflng()` | none |
| `ogetpi()` | `OCIStmtGetPieceInfo()` |
| `olog()` | `OCILogon()` or `OCILogon2()` |
| `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()` | `OCIStmtPrepare2()`; 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()` |

**Table 3-3 (Cont.) Obsolescent OCI Functions**

| *7.x* OCI Routine | Equivalent or Similar Later OCI Routine |
|---|---|
| `orlon()` | `OCILogon()` |
| `oname()` | Note: see `odescr()` preceding |
| `osql3()` | Note: see `oparse()` preceding |

**Note:**

Applications using `size_t` to define host area (`hda`) that is passed to the OCI 7 calls, may crash on some platforms due to misaligned data. `Hda_Def` should be used instead of `size_t` in that case. If the application cannot be modified, the compiler and linker flag `misalign` can be used on a SPARC platform. Refer to the demos and the following documentation for more information: Sun Studio 12: C++ User's Guide A.2.53 —misalign.

**See Also:**

- OCI Programming Basics for information about what additional program logic may be required before or after the new or current call is made.

## 3.7.5 OCI Routines Not Supported

Some OCI routines that were available in previous versions of OCI are not supported in the current release.

They are listed in Table 3-4.

**Table 3-4 OCI Functions Not Supported**

| OCI Routine | Equivalent or Similar Later OCI Routine |
|---|---|
| `obind()` | `OCIBindByName()`, `OCIBindByPos()` (Note: additional bind calls may be necessary for some data types) |
| `obindn()` | `OCIBindByName()`, `OCIBindByPos()` (Note: additional bind calls may be necessary for some data types) |
| `odfinn()` | `OCIDefineByPos()` (Note: additional define calls may be necessary for some data types) |
| `odsrbn()` | Note: see `odescr()` in Table 3-3 |
| `ologon()` | `OCILogon()` |
| `osql()` | Note: see `oparse()` Table 3-3 |

# 3.8 Fault Diagnosability in OCI

This section describes the following topics:

- About Fault Diagnosability in OCI
- ADR Base Location
- Using ADRCI
- Controlling ADR Creation and Disabling Fault Diagnosability Using sqlnet.ora

## 3.8.1 About Fault Diagnosability in OCI

Fault diagnosability was introduced into OCI in Oracle Database 11*g* Release 1 (11.1).

An incident (an occurrence of a problem) on the OCI client is captured without user intervention in the form of diagnostic data: dump files or core dump files. Up to Release 11.2.0.1, the diagnostic data was stored in an Automatic Diagnostic Repository (ADR) subdirectory created for the incident. For example, if a Linux or UNIX application fails with a `NULL` pointer reference, then the core file is written in the ADR home directory (if it exists) instead of the operating system directory. The ADR subdirectory structure and a utility to deal with the output, ADR Command Interpreter (ADRCI), are discussed in the following sections. However, beginning with Release 11.2.0.2, the diagnostic data is stored in the current directory.

An ADR home is the root directory for all diagnostic data for an instance of a particular product such as OCI and a particular operating system user. ADR homes are grouped under the same root directory, the ADR base.

Fault diagnosability and the ADR structure for Oracle Database are described in detail in the discussion of managing diagnostic data in Oracle Database Administrator's Guide.

## 3.8.2 ADR Base Location

Describes how the location of the ADR base is determined.

The location of the ADR base is determined by OCI in the following order:

1. For all diagnosability parameters, OCI first looks in the file `oraaccess.xml`. If these parameters are not set there, then OCI looks next in `sqlnet.ora` (if it exists) for a statement such as (Linux or UNIX):

   ```
   ADR_BASE=/foo/adr
   ```

   `adr` (the name of a directory) must exist and be writable by all operating system users who execute OCI applications and want to share the same ADR base. `foo` stands for a path name. The location of `sqlnet.ora` is given in the directory `$TNS_ADMIN` (`%TNS_ADMIN%` on Windows). If there is no `$TNS_ADMIN` then the current directory is used. If `ADR_BASE` is set and one `sqlnet.ora` is shared by all users, then OCI stops searching when directory `adr` does not exist or is not writable by the user. If `ADR_BASE` is not set, then OCI continues the search, testing for the existence of certain directories.

   For example, if `sqlnet.ora` contains the entry `ADR_BASE=/home/chuck/test` then the ADR base is `/home/chuck/test/oradiag_chuck` and the ADR home could be something like `/home/chuck/test/oradiag_chuck/diag/clients/user_chuck/host_4144260688_11`.

2. *`$ORACLE_BASE`* (*`%ORACLE_BASE%`* on Windows) exists. In this case, the client subdirectory exists because it was created during installation of the database using Oracle Universal Installer.

   For example, if *`$ORACLE_BASE`* is `/home/chuck/obase` then the ADR base is `/home/chuck/obase` and the ADR home could be similar to `/home/chuck/obase/diag/clients/user_chuck/host_4144260688_11`.

3. *`$ORACLE_HOME (%ORACLE_BASE%`* on Windows) exists. In this case, the client subdirectory exists because it was created during installation of the database using Oracle Universal Installer.

   For example, if *`$ORACLE_HOME`* is `/ade/chuck_l1/oracle` then the ADR base is `/ade/chuck_l1/oracle/log` and the ADR home could be similar to `/ade/chuck_l1/oracle/log/diag/clients/user_chuck/host_4144260688_11`.

4. Operating system home directory: *`$HOME`* on Linux or UNIX, or *`%USERPROFILE%`* on Windows. On Linux or UNIX the location could be something like this for user `chuck`: `/home/chuck/oradiag_chuck`. On Windows, a folder named `Oracle` is created under `C:\Documents and Settings\chuck`.

   For example, in an Instant Client, if *`$HOME`* is `/home/chuck` then the ADR base is `/home/chuck/oradiag_chuck` and the ADR home could be `/home/chuck/oradiag_chuck/diag/clients/user_chuck/host_4144260688_11`.

5. On Windows, if the application is run as a service, the home directory option is skipped.

6. Temporary directory in the Linux or UNIX operating system: `/var/tmp`.

   For example, in an Instant Client, if *`$HOME`* is not writable, then the ADR base is `/var/tmp/oradiag_chuck` and the ADR home could be `/var/tmp/oradiag_chuck/diag/clients/user_chuck/host_4144260688_11`.

   Temporary directories in the Windows operating system, searched in this order:

   a. *`%TMP%`*
   b. *`%TEMP%`*
   c. *`%USERPROFILE%`*
   d. Windows system directory

If none of these directory choices are available and writable, or the ADR base is not created and there are no diagnostics.

**See Also:**

- About Oracle Instant Client
- *Oracle Database Net Services Reference*

## 3.8.3 Using ADRCI

ADRCI is a command-line tool that enables you to view diagnostic data within the ADR and to package incident and problem information into a zip file for Oracle Support to use.

You can use ADRCI interactively and from a script. A *problem* is a critical error in OCI or the client. Each problem has a problem key. An *incident* is a single occurrence of a problem and is identified by a unique numeric incident ID. Each incident has a problem key that is a set of attributes: the `ORA` error number, error parameter values, and other information. Two incidents have the same root cause if their problem keys match.

What follows is a launch of ADRCI in a Linux system, a use of the `HELP` command for the `SHOW BASE` command, and then the use of the `SHOW BASE` command with the option `-PRODUCT CLIENT`, which is necessary for OCI applications. The ADRCI commands are case-insensitive. User input is shown in bold.

```
% adrci

ADRCI: Release 12.2.0.0.0 - Development on Wed Dec 2 18:26:29 2015
Copyright (c) 1982, 2015, Oracle.  All rights reserved. ADR base = "/u01/app/oracle/
log"


adrci> help show base

  Usage: SHOW BASE [-product <product_name>]

  Purpose: Show the current ADR base setting.

  Options:
    [-product <product_name>]: This option allows users to show the
    given product's ADR Base location. The current registered products are
    "CLIENT" and "ADRCI".

  Examples:
    show base -product client
    show base

adrci> show base -product client
ADR base = "/u01/app/oracle/log"
```

Next, the `SET BASE` command is described:

```
adrci> help set base

  Usage:  SET BASE <base_str> | -product <product_name>

  Purpose: Set the ADR base to use in the current ADRCI session.
           If there are valid ADR homes under the base, all homes will
           will be added to the current ADRCI session.

  Arguments:
    <base_str>: It is the ADR base directory, which is a system-dependent
    directory path string.
    -product <product_name>: This option allows users to set the
    given product's ADR Base location. The current registered products are
    "CLIENT" and "ADRCI".
  Notes:
    On platforms that use "." to signify current working directory,
    it can be used as base_str.

  Example:
    set base /net/sttttd1/scratch/someone/view_storage/someone_v1/log
    set base -product client
    set base .

adrci> quit
```

When `ADRCI` is started, the default ADR base is for the `rdbms` server. *`$ORACLE_HOME`* is set to `"/u01/app/oracle/"`:

```
% adrci

ADRCI: Release 12.2.0.0.0 - Development on Wed Dec 2 18:26:29 2015

Copyright (c) 1982, 2015, Oracle.  All rights reserved.

ADR base = "/u01/app/oracle/log"
 adrci>
```

For OCI application incidents you must check and set the base:

```
adrci> show base -product client
ADR base is "/u01/app/oracle/log"
adrci> set base /home/chuck_13/oradiag_chuck
```

For Instant Client there is no *`$ORACLE_HOME`*, so the default base is the user's home directory:

```
adrci> show base -product client
ADR base is "/home/chuck_13/oradiag_chuck"
adrci> set base /home/chuck/oradiag_chuck
adrci> show incidents

ADR Home = /home/chuck/oradiag_chuck/diag/clients/user_chuck/host_4144260688_11:
*************************************************************************
INCIDENT_ID    PROBLEM_KEY           CREATE_TIME
-------------------------------------------------------------------------
1                      oci 24550 [6]             2015-12-02 17:20:02.803697
-07:00
1 rows fetched

adrci>
```

**See Also:**

About Oracle Instant Client

*Oracle Database Utilities* for an introduction to ADRCI

## 3.8.4 Controlling ADR Creation and Disabling Fault Diagnosability Using sqlnet.ora

Describes how to control ADR creation and disabling fault diagnosability using sqlnet.ora.

To disable diagnosability, turn off diagnostics by setting the following parameters in `sqlnet.ora` (the default is `TRUE`):

```
DIAG_ADR_ENABLED=FALSE
DIAG_DDE_ENABLED=FALSE
```

To turn off the OCI signal handler and reenable standard operating system failure processing, place the following parameter setting in `sqlnet.ora`:

```
DIAG_SIGHANDLER_ENABLED=FALSE
```

As noted previously, `ADR_BASE` is used in `sqlnet.ora` to set the location of the ADR base.

Oracle Database client contains advanced features for diagnosing issues, including the ability to dump diagnostic information when important errors are detected. By default, these dumps are restricted to a small subset of available information, to ensure that application data is not dumped. However, in many installations, secure locations for dump files may be configured, ensuring the privacy of such logs. In such cases, it is recommended to turn on full dumps; this can greatly speed resolution of issues. Full dumps can be enabled by adding the following line to the `sqlnet.ora` file used by your Oracle Database client installation:

```
DIAG_RESTRICTED=FALSE
```

To verify that diagnosability features are working correctly:

1. Upgrade your application to use the latest client libraries.
2. Start your application.
3. Check the file `sqlnet.log` in your application's `TNS_ADMIN` directory for error messages indicating that diagnosability could not be started (normally this is due to invalid directory names or permissions).

**See Also:**

- *Oracle Database Net Services Reference* for the ADR parameter settings in `sqlnet.ora`
- *Oracle Database Net Services Administrator's Guide* for more information about the structure of ADR

# 4

# OCI Programming Basics

This chapter introduces concepts and procedures involved in programming with OCI.

After reading this chapter, you should have most of the tools necessary to understand and create a basic OCI application.

This chapter includes the following major sections:

- Overview of OCI Program Programming
- OCI Data Structures
- OCI Programming Steps
- Error Handling in OCI
- Additional Coding Guidelines
- About Using PL/SQL in an OCI Program
- OCI Globalization Support

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:**
>
> - *Oracle Database Globalization Support Guide* for a discussion of the OCI functions that apply to a multilingual environment
> - *Oracle Database Data Cartridge Developer's Guide* for a discussion of the OCI functions that apply to cartridge services

## 4.1 Overview of OCI Program Programming

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.

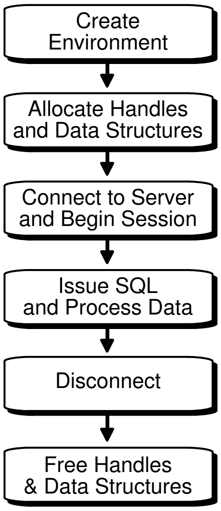
OCI uses the following basic program flow:

1. Create the environment by initializing 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 disconnect from server connections.
6. Free handles and data structures.

Figure 4-1 illustrates the flow of steps in an OCI application. OCI Programming Steps describes each step in more detail.

**Figure 4-1 Basic OCI Program Flow**


Create Environment
Allocate Handles and Data Structures
Connect to Server and Begin Session
Issue SQL and Process Data
Disconnect
Free Handles & Data Structures

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 multiple active connections and statements in an OCI application.

> **See Also:**
>
> OCI Object-Relational Programming through Using the Object Type Translator with OCI for information about accessing and manipulating objects

# 4.2 OCI Data Structures

*Handles* and *descriptors* are opaque data structures that are defined in OCI applications.

Handles and descriptors can be allocated directly, through specific allocate calls, or they can be implicitly allocated by OCI functions.

> **Note:**
>
> Programmers who have previously written 7.x OCI applications must become familiar with these data structures that 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.

This section includes the following topics: Handles

**Related Topics**

- OCI Descriptors
  OCI *descriptors* and *locators* are opaque data structures that maintain data-specific information.

## 4.2.1 Handles

Almost every OCI call includes in its 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 must access the information stored in handles. The get and set attribute OCI calls, `OCIAttrGet()` and `OCIAttrSet()`, access and set this information.

Table 4-1 lists the handles defined for OCI. For each handle type, the C data type and handle type constant used to identify the handle type in OCI calls are listed.

**Table 4-1 OCI Handle Types**

| Description | C Data Type | 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` |

**Table 4-1 (Cont.) OCI Handle Types**

| Description | C Data Type | Handle Type Constant |
|---|---|---|
| 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` | Not applicable |
| OCI subscription handle | `OCISubscription` | `OCI_HTYPE_SUBSCRIPTION` |
| OCI direct path context handle | `OCIDirPathCtx` | `OCI_HTYPE_DIRPATH_CTX` |
| OCI direct path function context handle | `OCIDirPathFuncCtx` | `OCI_HTYPE_DIRPATH_FN_CTX` |
| OCI direct path column array handle | `OCIDirPathColArray` | `OCI_HTYPE_DIRPATH_COLUMN_ARRAY` |
| OCI direct path stream handle | `OCIDirPathStream` | `OCI_HTYPE_DIRPATH_STREAM` |
| OCI process handle | `OCIProcess` | `OCI_HTYPE_PROC` |
| OCI administration handle | `OCIAdmin` | `OCI_HTYPE_ADMIN` |
| OCI HA event handle | `OCIEvent` | Not applicable |
| OCI SODA collection handle | `OCISodaColl` | `OCI_HTYPE_SODA_COLLECTION` |
| OCI SODA collection cursor handle | `OCISodaCollCursor` | `OCI_HTYPE_SODA_CURSOR` |
| OCI SODA document cursor handle | `OCISodaDocCursor` | `OCI_HTYPE_SODA_DOC_CURSOR` |
| OCI SODA document handle | `OCISodaDoc` | `OCI_HTYPE_SODA_DOCUMENT` |
| OCI SODA output options handle | `OCISodaOutputOptions` | `OCI_HTYPE_SODA_OUTPUT_OPTIONS` |
| OCI SODA operation options handle | `OCISodaOperationOptions` | `OCI_HTYPE_SODA_OPER_OPTIONS` |

This section includes the following topics:

- About Allocating and Freeing Handles
- Environment Handle
- Error Handle
- Service Context Handle and Associated Handles
- Statement, Bind, and Define Handles
- Describe Handle
- Complex Object Retrieval Handle
- Thread Handle
- Subscription Handle

- Direct Path Handles
- Connection Pool Handle
- Handle Attributes
- OCI Descriptors

**Related Topics**

- Handle Attributes
- OCIAttrSet()
- OCIAttrGet()

### 4.2.1.1 About Allocating and Freeing Handles

Your application allocates all handles (except the bind, define, and thread handles) for 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 for 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()`.

The types of handles include: session pool 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.

**Related Topics**

- OCI Demonstration Programs
- OCIEnvCreate()
- OCIEnvNlsCreate()
- OCIHandleAlloc()
- OCIThreadHndInit()
- OCIHandleFree()

### 4.2.1.2 Environment Handle

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

### 4.2.1.3 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 a parameter.

**Related Topics**

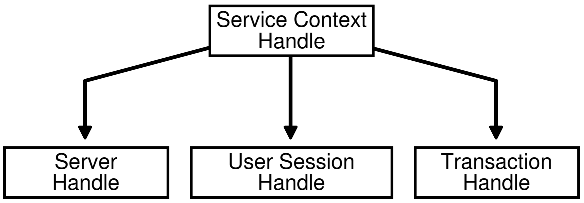
- About Implementing Thread Safety
  To take advantage of thread safety, an application must be running on a thread-safe operating system.

### 4.2.1.4 Service Context Handle and Associated Handles

A *service context handle* defines attributes that determine the operational context for OCI calls to a server.

The service context handle contains three handles as its attributes, that represent a server connection, a user session, and a transaction. These attributes are illustrated in Figure 4-2.

**Figure 4-2    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 handle down in this way provides scalability and enables programmers to create sophisticated multitiered applications and transaction processing (TP) monitors to execute requests on behalf of multiple users on multiple application servers and different transaction contexts.

You must allocate and initialize the service context handle with `OCIHandleAlloc()`, `OCILogon()`, or `OCILogon2()` before you can use it. The service context handle is allocated explicitly by `OCIHandleAlloc()`. It can be initialized using `OCIAttrSet()` with the server, user 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 handle must be explicitly allocated, and the server and user session handles must be explicitly set into the service context handle. `OCIServerAttach()` and `OCISessionBegin()` calls initialize the server and user session handle respectively.

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

**See Also:**

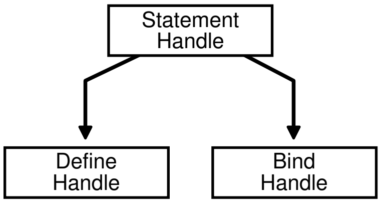
- OCI Support for Transactions
- OCI Environment Initialization, and Password and Session Management for more information about establishing a server connection and user session
- OCIHandleAlloc()
- OCILogon()
- OCILogon2()
- OCIAttrSet()
- OCIServerAttach()
- OCISessionBegin()
- OCISessionGet()

### 4.2.1.5 Statement, Bind, and Define Handles

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

A statement handle is shown in Figure 4-3.

**Figure 4-3 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 `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()` function. The user must not 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()` or `OCIDefineByPos2()`. The user must not allocate define handles. They are implicitly allocated by the define call.

Bind and define handles are implicitly allocated by the OCI library, and are transparently reused if the bind or define operation is repeated. The actual value of the bind or define handle is needed by the application for the advanced bind or define operations described in Binding and Defining in OCI. The handles are freed when the statement handle is freed or when a new statement is prepared on the statement handle. Explicitly allocating bind or define handles may lead to memory leaks. Explicitly freeing bind or define handles may cause abnormal program termination.

> **See Also:**
>
> - "Advanced Bind Operations in OCI"
> - "Advanced Define Operations in OCI"
> - OCIBindByName()
> - OCIBindByName2()
> - OCIBindByPos()
> - OCIBindByPos2()
> - OCIDefineByPos()
> - OCIDefineByPos2()

### 4.2.1.6 Describe Handle

The *describe handle* is used by the OCI describe call, `OCIDescribeAny()`.

The `OCIDescribeAny()` call obtains information about schema objects in a database (for example, functions or 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 the parameter descriptors.

> **See Also:**
>
> - Describing Schema Metadata for more information about using the `OCIDescribeAny()` function
> - OCIDescribeAny()

### 4.2.1.7 Complex Object Retrieval Handle

The *complex object retrieval (COR) handle* is used by some OCI applications that work with objects in an Oracle database.

The complex object retrieval (COR) handle contains *COR descriptors,* provides instructions for retrieving objects referenced by another object.

**Related Topics**

- Complex Object Retrieval
  A complex object includes its root object and its set of logically related objects each of which are prefetched based on a given depth level.

### 4.2.1.8 Thread Handle

The thread handle is used in multithreaded applications.

For information about the thread handle, which is used in multithreaded applications, see the following OCIThread Package.

**Related Topics**

- OCIThread Package
  The `OCIThread` package provides some commonly used threading primitives.

### 4.2.1.9 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.

**Related Topics**

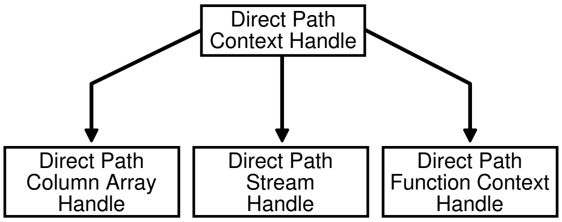
- Publish-Subscribe Notification in OCI
  The publish-subscribe notification feature allows an OCI application to receive client notifications directly, register an email address to which notifications can be sent, register an HTTP URL to which notifications can be posted, or register a PL/SQL procedure to be invoked on a notification.

### 4.2.1.10 Direct Path Handles

The direct path handles are necessary for an OCI application that uses the direct path load engine in the Oracle database.

The direct path load interface enables the application to access the direct block formatter of the Oracle database. Figure 4-4 shows the different kinds of direct path handles.

**Figure 4-4 Direct Path Handles**


Direct Path Context Handle
Direct Path Column Array Handle
Direct Path Stream Handle
Direct Path Function Context Handle

**Related Topics**

- Direct Path Loading Overview
  The direct path load interface enables an OCI application to access the direct path load engine of Oracle Database to perform the functions of the SQL*Loader utility.
- Direct Path Loading Handle Attributes

### 4.2.1.11 Connection Pool Handle

The *connection pool handle* is used for applications that pool physical connections into virtual connections.

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

**Related Topics**

- Connection Pooling in OCI
  Connection pooling is the use of a group (the pool) of reusable physical connections by several sessions to balance loads.

### 4.2.1.12 Handle Attributes

All OCI handles have *attributes* that represent data stored in that handle.

You can read handle attributes by using the attribute get call, `OCIAttrGet()`, and you can change them with the attribute set call, `OCIAttrSet()`.

For example, the statements in Example 4-1 set the user name in the session handle by writing to the `OCI_ATTR_USERNAME` attribute:

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, as shown in Example 4-2.

**See Also:**

- The description of OCIArrayDescriptorAlloc() for an example showing how to allocate a large number of descriptors
- Handle and Descriptor Attributes
- OCIAttrGet()
- OCIAttrSet()
- OCISessionBegin()
- OCIStmtExecute()

**Example 4-1 Using the OCI_ATTR_USERNAME Attribute to Set the User Name in the Session Handle**

```
text username[] = "hr";
err = OCIAttrSet ((void *) mysessp, OCI_HTYPE_SESSION, (void *)username,
      (ub4) strlen((char *)username), OCI_ATTR_USERNAME, (OCIError *) myerrhp);
```

**Example 4-2 Returning Describe Information in the Statement Handle Relating to Select-List Items**

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

## 4.2.1.13 OCI Descriptors

OCI *descriptors* and *locators* are opaque data structures that maintain data-specific information.

Table 4-2 lists OCI descriptors, along with their C data type, 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 4-2 Descriptor Types**

| Description | C Data Type | OCI Type Constant |
|---|---|---|
| Snapshot descriptor | `OCISnapshot` | `OCI_DTYPE_SNAP` |
| Result set descriptor | `OCIResult` | `OCI_DTYPE_RSET` (Deprecated) |
| LOB data type locator | `OCILobLocator` | `OCI_DTYPE_LOB` |
| `BFILE` data type 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` |
| User callback descriptor | `OCIUcb` | `OCI_DTYPE_UCB` |
| 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 4-2 (Cont.) Descriptor Types

| Description | C Data Type | OCI Type Constant |
|---|---|---|
| Table change | None | `OCI_DTYPE_TABLE_CHDES` |
| Row change | None | `OCI_DTYPE_ROW_CHDES` |
| Shard key and shard group key descriptor | `OCIShardkey` | `OCI_DTYPE_SHARD_KEY` |

> **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. LOB and BFILE Locators discusses this difference.

The following list describes the main purpose of each descriptor type. The sections that follow describe each descriptor type in more detail:

- `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 data types
- `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

This section includes the following topics:

- Snapshot Descriptor
- LOB and BFILE Locators
- Parameter Descriptor
- ROWID Descriptor
- Date, Datetime, and Interval Descriptors
- Complex Object Descriptor
- Advanced Queuing Descriptors
- User Memory Allocation

**Related Topics**

- OCIDescriptorAlloc()
- OCIDescriptorFree()

- OCIArrayDescriptorAlloc()
- OCIArrayDescriptorFree()
  Free a previously allocated array of descriptors.

### 4.2.1.13.1 Snapshot Descriptor

The *snapshot descriptor* is an optional parameter to the execute call, `OCIStmtExecute()`.

The snapshot descriptor indicates that a query is being executed against a database snapshot that represents the state of a database at a particular time.

Allocate a snapshot descriptor with a call to `OCIDescriptorAlloc()` by passing `OCI_DTYPE_SNAP` as the `type` parameter.

**Related Topics**

- OCIStmtExecute()
- OCIDescriptorAlloc()
- 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.

### 4.2.1.13.2 LOB and BFILE Locators

A large object (LOB) is an Oracle data type that can hold binary large object (`BLOB`) or character large object (`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.

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.

> **Note:**
>
> 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 by 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 Oracle database 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.

**Related Topics**

- LOB and BFILE Operations
  This chapter describes LOB and BFILE operations.
- About Binding LOB Data
  Oracle Database allows nonzero binds for `INSERT`s and `UPDATE`s of any size LOB.
- About Defining LOB Data
- OCIDescriptorAlloc()

### 4.2.1.13.3 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.

**Related Topics**

- OCIDescriptorAlloc()
- OCIParamGet()
- About 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.
- Describing Schema Metadata

### 4.2.1.13.4 ROWID Descriptor

The `ROWID` descriptor, `OCIRowid`, is used by applications that must retrieve and use Oracle ROWIDs.

To work with a `ROWID` 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 operation.

**Related Topics**

- OCIAttrGet()

### 4.2.1.13.5 Date, Datetime, and Interval Descriptors

The date, datetime, and interval descriptors are used by applications that use the date, datetime, or interval data types (`OCIDate`, `OCIDateTime`, and `OCIInterval`).

These descriptors can be used for binding and defining, and are passed as parameters to the functions `OCIDescriptorAlloc()` and `OCIDescriptorFree()` to allocate and free memory.

**Related Topics**

- OCIDescriptorAlloc()
- OCIDescriptorFree()
- Data Types
- OCI Data Type Mapping and Manipulation Functions

### 4.2.1.13.6 Complex Object Descriptor

Complex object retrieval (COR) may improve application performance when dealing with objects.

Application performance when dealing with objects may be increased using *complex object retrieval (COR)*.

**Related Topics**

- Complex Object Retrieval
  A complex object includes its root object and its set of logically related objects each of which are prefetched based on a given depth level.

### 4.2.1.13.7 Advanced Queuing Descriptors

There are a number of Oracle Database Advanced Queuing descriptors for use to maintain data-specific information.

Oracle Database Advanced Queuing provides message queuing as an integrated part of Oracle Database.

**Related Topics**

- OCI and Database Advanced Queuing
  OCI provides an interface to the Database Advanced Queuing (Database AQ) feature. Database Advanced Queuing provides message queuing as an integrated part of Oracle Database.
- Publish-Subscribe Registration Functions in OCI
  You can register directly to the database or register using Lightweight Directory Access Protocol (LDAP).

#### 4.2.1.13.8 User Memory Allocation

The `OCIDescriptorAlloc()` call has an `xtramem_sz` parameter in its parameter list.

The `xtramem_sz` parameter is used to specify the amount of user memory that 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 can be 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 that has the same lifetime as the handle.

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

**Related Topics**

- OCIDescriptorAlloc()
- OCIDescriptorFree()
- OCIHandleAlloc()
- OCIEnvCreate()
- OCIEnvInit()

## 4.3 OCI Programming Steps

The following sections describe in detail each of the steps in developing an OCI application.

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

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

- OCI Environment Initialization
- About Processing SQL Statements in OCI
- Commit or Roll Back Operations
- About Terminating the Application
- Error Handling in OCI

**See Also:**

- The first sample program in OCI Demonstration Programs for an example showing the use of OCI calls for processing SQL statements.
- Runtime Data Allocation and Piecewise Operations in OCI for a detailed description of the special case of dynamically providing data at run time
- About Binding and Defining Arrays of Structures in OCI for a description of the special considerations for operations involving arrays of structures
- Error Handling in OCI for an outline of the steps involved in processing a SQL statement within an OCI program
- Overview of OCI Multithreaded Development for information about using the OCI to write multithreaded applications
- SQL Statements for more information about types of SQL statements

## 4.3.1 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:

- About Creating the OCI Environment
- About Allocating Handles and Descriptors
- Application Initialization, Connection, and Session Creation

### 4.3.1.1 About Creating the OCI Environment

Each OCI function call is executed in the context of an environment that is created with the `OCIEnvCreate()` call.

The `OCIEnvCreate()` 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 can:

- Run in a threaded environment (`mode` = `OCI_THREADED`).
- Use objects (`mode` = `OCI_OBJECT`). Use with AQ subscription registration.
- 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(), OCIEnvNlsCreate(), and OCIInitialize() (deprecated) for more information about the initialization calls
> - Overview of OCI Multithreaded Development
> - OCI Object-Relational Programming , Object-Relational Data Types in OCI, Direct Path Load Interface, Object Advanced Topics in OCI, and Using the Object Type Translator with OCI
> - Publish-Subscribe Notification in OCI

### 4.3.1.2 About Allocating Handles and Descriptors

Oracle Database 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()` or `OCIBindByPos2()`, allocates the handles for you.

You can allocate the types of handles listed in Table 4-1 with `OCIHandleAlloc()` Depending on the functionality of your application, it must allocate some or all of these handles.

**Related Topics**

- OCIHandleAlloc()
- OCIBindByPos()
- OCIBindByPos2()

### 4.3.1.3 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 an Oracle database connection and beginning a user session.

These methods include:

- Single User, Single Connection
- Client Access Through a Proxy
- Nonproxy Multiple Sessions or Connections

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

#### 4.3.1.3.1 Single User, Single Connection

The single user, single connection 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()` or `OCILogon()`, the OCI library initializes the service context handle that is passed to it, and creates a connection to the specified Oracle database for the user making the request.

Example 4-3 shows what a call to `OCILogon2()` looks like for a single user session with user name `hr`, password `hr`, and database `oracledb`.

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 are all read-only; the application cannot switch session or transaction by changing the appropriate attributes of the service context handle using an `OCIAttrSet()` call.

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

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

**Example 4-3 Using the OCILogon2 Call for a Single User Session**

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

**See Also:**

Operating System Considerations for information regarding operating systems providing facilities for spawning processes that allow child processes to reuse state created by their parent process. This section explains why the child process must not use the same database connection as created by the parent.

### 4.3.1.3.2 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 in to the database as a proxy user. A proxy user can switch identities and, after logging in to 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.

**Note:**

In release 1 of Oracle 11*g*, standards for acceptable passwords were greatly raised to increase security. Examples of passwords in this section are incorrect. A password must contain no fewer than eight characters. See the guidelines for securing passwords *Oracle Database Security Guide* for additional information.

Proxy to database users is supported by using 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 afterward. In OCI, you can either use connect strings or the function `OCIAttrSet()` with the parameter `OCI_ATTR_PROXY_CLIENT`.

Even though beginning with Oracle Database 12c Release 2 (12.2) the maximum length of each identifier is increased to 128 bytes, the user name and proxy combination can not exceed 250 bytes.

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.

Example 4-4 through Example 4-11 show connect strings that you can use 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.

In Example 4-4, Dilbert and Joe are two local database users. To enable Dilbert to serve as a proxy for Joe, use the SQL statement shown in Example 4-4.

When user name `dilbert` is acting on behalf of `joe`, use the connection string shown in Example 4-5. (The user name `dilbert` has the password `tiger123`).

The left and right brackets "[" and "]" are entered in the connection string.

In Example 4-6, "Dilbert" and "Joe" are two local database users. The names are case-sensitive and must be enclosed in double quotation marks. To enable "Dilbert" to serve as a proxy for "Joe", use the SQL statement shown in Example 4-6.

When "Dilbert" is acting on behalf of "Joe", use the connection string shown in Example 4-7. Be sure to include the double quotation marks (") characters.

When the proxy user is created as "dilbert[mybert]", use the connection string shown in Example 4-8 to connect to the database. (The left and right brackets "[" and "]" are entered in the connection string.)

In Example 4-9, dilbert[mybert] and joe[myjoe] are two database users that contain the left and right bracket characters "[" and "]". If dilbert[mybert] wants to act on behalf of joe[myjoe], Example 4-9 shows the connect statement to use.

In Example 4-10, you can set the target user name by using the `ALTER USER` statement.

**See Also:**

- OCI_ATTR_PROXY_CLIENT
- *Oracle Database Security Guide* for a discussion of proxy authentication
- Password and Session Management
- OCIAttrSet()

**Note:**

There are compatibility issues of client access through a proxy. Because this feature was introduced in Oracle Database release 10.2, pre-10.2 clients do not have it. If newer clients use the feature with pre-10.2 Oracle databases, the connect fails and the client returns an error after checking the database release level.

#### Example 4-4 Enabling a Local User to Serve as a Proxy for Another User

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

#### Example 4-5 Connection String to Use for the Proxy User

```
dilbert[joe]/tiger123@db1
```

#### Example 4-6 Preserving Case Sensitivity When Enabling a Local User to Serve as a Proxy for Another User

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

### Example 4-7 Preserving Case Sensitivity in the Connection String

```
"Dilbert"["Joe"]/tiger123@db1
```

### Example 4-8 Using "dilbert[mybert]" in the Connection String

```
"dilbert[mybert]"/tiger123

rem the user was already created this way:
rem CREATE USER "dilbert[mybert]" IDENTIFIED BY tiger123;
```

### Example 4-9 Using "dilbert[mybert]"["joe[myjoe]"] in the Connection String

```
"dilbert[mybert]"["joe[myjoe]"]/tiger123
```

### Example 4-10 Setting the Target User Name

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

Then, as shown in Example 4-11, in an OCI program, use the `OCIAttrSet()` call to set the attribute `OCI_ATTR_PROXY_CLIENT` and the proxy `dilbert`. In your program, use these statements to connect multiple times.

### Example 4-11 Using OCI to Set the OCI_ATTR_PROXY_CLIENT Attribute and the Proxy dilbert

```
OCIAttrSet(session, OCI_HTYPE_SESSION, (void  *)"dilbert",
           (ub4)strlen("dilbert"), OCI_ATTR_USERNAME,
           error_handle);
OCIAttrSet(session, OCI_HTYPE_SESSION, (void  *)"tiger123",
           (ub4)strlen("tiger123"), OCI_ATTR_PASSWORD,
           error_handle);
OCIAttrSet(session, OCI_HTYPE_SESSION, (void  *)"joe",
           (ub4)strlen("joe"), OCI_ATTR_PROXY_CLIENT,
           error_handle);
```

## 4.3.1.3.3 Nonproxy Multiple Sessions or Connections

The nonproxy multiple sessions or connections 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 Oracle database and begin sessions are:

- `OCIServerAttach()` - Creates an access path to the Oracle database for OCI operations.
- `OCISessionBegin()` - Establishes a session for a user against a particular Oracle database. This call is required for the user to execute operations on the Oracle database.

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 nonmigratable 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
- Session Pooling and Connection Pooling in OCI for more information about maintaining multiple sessions, transactions, and connections
- Client Character Set Control from OCI for the use of `OCIEnvNlsCreate()`

Example 4-12 demonstrates the creation and initialization of 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 simplicity, error checking is not included.

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

**Example 4-12    Creating and Initializing an OCI Environment**

```
#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, (void  *)0,
                    0, 0, 0, (size_t) 0, (void  **)0);

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

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

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

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

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

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

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

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

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

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

## 4.3.2 About Processing SQL Statements in OCI

What are the specific steps involved in processing SQL statements in OCI.

Using SQL Statements in OCI outlines the specific steps involved in processing SQL statements in OCI.

## 4.3.3 Commit or Roll Back Operations

An application commits changes to the database by calling `OCITransCommit()`.

The `OCITransCommit()` 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:**
>
> By using the `OCI_COMMIT_ON_SUCCESS` mode of the `OCIStmtExecute()` call, the application can selectively commit transactions after each statement execution, saving an extra round-trip.

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

If an application disconnects from Oracle Database in a 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 Handle and Associated Handles"
- "OCI Support for Transactions"

### 4.3.4 About Terminating the Application

What should an application do before it terminates.

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

1. Delete the user session by calling `OCISessionEnd()` for each session.
2. Delete access to the data sources 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:**

The first sample program in OCI Demonstration Programs for an example showing handles being freed at the end of an application

**Note:**

If the application uses the simplified logon method of `OCILogon2()`, then a call to `OCILogoff()` terminates the session, disconnects from the Oracle database, and frees the service context and associated handles. The application is still responsible for freeing other handles it allocated.

## 4.3.5 Error Handling in OCI

OCI function calls have a set of return codes.

These OCI function call return codes are listed in Table 4-3, 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:**

- "Functions Returning Other Values" for exceptions
- "OCIErrorGet()" for complete details and an example of usage
- "Server Handle Attributes"

**Table 4-3 OCI Return Codes**

| OCI Return Code | Value | Description |
|---|---|---|
| `OCI_SUCCESS` | 0 | The function completed successfully. |
| `OCI_SUCCESS_WITH_INFO` | 1 | The function completed successfully; a call to `OCIErrorGet()` returns additional diagnostic information. This may include warnings. |
| `OCI_NO_DATA` | 100 | The function completed, and there is no further data. |
| `OCI_ERROR` | -1 | The function failed; a call to `OCIErrorGet()` returns additional information. |
| `OCI_INVALID_HANDLE` | -2 | An invalid handle was passed as a parameter or a user callback was passed an invalid handle or invalid context. No further diagnostics are available. |
| `OCI_NEED_DATA` | 99 | The application must provide runtime data. |
| `OCI_STILL_EXECUTING` | -3123 | 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` | -24200 | This code is returned only from a callback function. It indicates that the callback function wants the OCI library to resume its normal processing. |
| `OCI_ROWCBK_DONE` | -24201 | This code is returned only from a callback function. It indicates that the callback function is done with the user row callback. |

If the return code indicates that an error has occurred, the application can retrieve error codes and messages specific to Oracle Database 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.

This section includes the following topics:

- Return and Error Codes for Data
- Functions Returning Other Values

### 4.3.5.1 Return and Error Codes for Data

The OCI return code, error number, indicator variable, and column return code are specified when the data fetched is normal, null, or truncated.

In Table 4-4, 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"

**Table 4-4 Return and Error Codes**

| State of Data | Return Code | Indicator - Not provided | Indicator - Provided |
|---|---|---|---|
| Not null or truncated | Not provided | `OCI_SUCCESS`<br>Error = 0 | `OCI_SUCCESS`<br>Error = 0<br>Indicator = 0 |
| Not null or truncated | Provided | `OCI_SUCCESS`<br>Error = 0<br>Return code = 0 | `OCI_SUCCESS`<br>Error = 0<br>Indicator = 0<br>Return code = 0 |
| Null data | Not provided | `OCI_ERROR`<br>Error = 1405 | `OCI_SUCCESS`<br>Error = 0<br>Indicator = -1 |
| Null data | Provided | `OCI_ERROR`<br>Error = 1405<br>Return code = 1405 | `OCI_SUCCESS`<br>Error = 0<br>Indicator = -1<br>Return code = 1405 |
| Truncated data | Not provided | `OCI_ERROR`<br>Error = 1406 | `OCI_ERROR`<br>Error = 1406<br>Indicator = data_len |

**Table 4-4 (Cont.) Return and Error Codes**

| State of Data | Return Code | Indicator - Not provided | Indicator - Provided |
|---|---|---|---|
| Truncated data | Provided | `OCI_SUCCESS_WITH_INFO`<br>Error = 24345<br>Return code = 1405 | `OCI_SUCCESS_WITH_INFO`<br>Error = 24345<br>Indicator = data_len<br>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.

### 4.3.5.2 Functions Returning Other Values

Some functions return values other than the OCI error codes.

These other OCI error codes are listed in Table 4-3. When you use these functions, 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.

# 4.4 Additional Coding Guidelines

This section explains some additional issues when coding OCI applications.

This section includes the following topics:

- Operating System Considerations
- Parameter Types
- Inserting Nulls into a Column
- Indicator Variables
- About Canceling Calls
- Positioned Updates and Deletes
- Reserved Words
- Polling Mode Operations in OCI
- Nonblocking Mode in OCI
- Setting Blocking Modes
- Canceling a Nonblocking Call

## 4.4.1 Operating System Considerations

Operating systems may provide facilities for spawning processes that allow child processes to reuse the state created by their parent process.

After spawning a child process, the child process must not use the same database connection as created by the parent. Any attempt on behalf of the child process to use the same database connection as the parent may cause undesired connection interference and result in intermittent `ORA-03137` errors, because Oracle Net expects only one user process to be using a connection to the database.

Where multiple, concurrent connections are required, consider using threads if your platform supports a threads package. Concurrent connections are supported in either single-threaded or multithreaded applications. For better performance with many concurrently opened connections, consider pooling them.

> **See Also:**
>
> - Overview of OCI Multithreaded Development
> - OCIThread Package
> - Session Pooling in OCI
> - When to Use Connection Pooling, Session Pooling, or Neither

## 4.4.2 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:

- Address Parameters
- Integer Parameters
- Character String Parameters

This section includes the following topics:

> **See Also:**
>
> Connect, Authorize, and Initialize Functions for more information about parameter data types and parameter passing conventions

### 4.4.2.1 Address Parameters

Address parameters are used to pass the address of the variable to Oracle Database.

You should be careful when developing in C, because it normally passes scalar parameters by value.

### 4.4.2.2 Integer Parameters

Binary integer and short binary integer parameters are numbers whose size is system-dependent.

See Oracle Database documentation that is specific to your operating system for the size of these integers on your system.

### 4.4.2.3 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.

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

## 4.4.3 Inserting Nulls into a Column

How to insert a null into a database column.

You can insert a null into a database column in several ways.

- One method is to use a literal `NULL` in the text of an `INSERT` or `UPDATE` statement. For example, the SQL statement makes the `ENAME` column `NULL`.

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

- Use indicator variables in the OCI bind call.
- Insert a `NULL` to set both the buffer length and maximum length parameters to zero on a bind call.

  > **Note:**
  >
  > Following the SQL standard requirements, Oracle Database 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.

**Related Topics**

- 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.

## 4.4.4 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 an Oracle database, the values of these indicator variables determine whether 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. For a `NULL` fetch in an `OCIStmtFetch2()` call or a truncation in an `OCIStmtExecute()` call, the OCI call returns `OCI_SUCCESS_WITH_INFO`. The output indicator variable is set.

The data type of indicator variables is `sb2`. For arrays of indicator variables, the individual array elements should be of type `sb2`.

This section includes the following topics:

- Input
- Output
- Indicator Variables for Named Data Types and REFs

## 4.4.4.1 Input

What values can be assigned to an indicator variable.

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

| Input Indicator Value | Action Taken by Oracle Database |
|---|---|
| -1 | Oracle Database assigns a `NULL` to the column, ignoring the value of the input variable. |
| >=0 | Oracle Database assigns the value of the input variable to the column. |

## 4.4.4.2 Output

What values can be assigned to an indicator variable.

On output, Oracle Database 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 Database 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. |

## 4.4.4.3 Indicator Variables for Named Data Types and REFs

Indicator variables for most data types introduced after release 8.0 behave as described earlier.

The only exception is SQLT_NTY (a named data type). 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 Using the Object Type Translator with OCI, and NULL Indicator Structure for information about NULL indicator structures
> - Descriptions of `OCIBindByName()` and `OCIBindByPos()` in Bind, Define, and Describe Functions, and the sections Information for Named Data Type and REF Binds, and Information for Named Data Type and REF Defines, and PL/SQL OUT Binds for more information about setting indicator parameters for named data types and `REF`s

## 4.4.5 About Canceling Calls

How do you cancel long-running or repeated OCI calls.

On most operating systems, you can cancel long-running or repeated OCI calls by entering the operating system's interrupt character (usually Control+C) from the keyboard.

> **Note:**
>
> This is not to be confused with canceling a cursor, which is accomplished by calling `OCIStmtFetch2()` 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.

When 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.

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

**See Also:**

- Overview of OCI Multithreaded Development
- OCIThread Package

## 4.4.6 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, consider a SQL statement such as the following:

```
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 ((void  *) envhp, (void  **) &rowid,
     (ub4) OCI_DTYPE_ROWID, (size_t) 0, (void  **) 0);
status = OCIAttrGet ((void *) mystmtp, OCI_HTYPE_STMT,
     (void *) 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 the following by binding the new salary to the `:1` placeholder and `rowid` to the `:2` placeholder.

```
UPDATE emp1 SET sal = :1 WHERE rowid = :2
```

Be sure to use data type code 104 (`ROWID` descriptor, see Table 5-2) when binding `rowid` to `:2`.

By using prefetching, you can select an array of `ROWID`s for use in subsequent batch updates.

**See Also:**

- UROWID and DATE for more information about `ROWID`s
- External Data Types for a table of external data types and codes

## 4.4.7 Reserved Words

Some words are reserved by Oracle.

That is, some reserved words 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.

This section includes the following topic: Oracle Reserved Namespaces

> **See Also:**
>
> *Oracle Database SQL Language Reference* and *Oracle Database PL/SQL Language Reference* to view the lists of the Oracle keywords or reserved words for SQL and PL/SQL

### 4.4.7.1 Oracle Reserved Namespaces

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

Table 4-5 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 4-5 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 |
| `OSN` | Oracle Net V1 |
| `TTC` | Oracle Net Two Task |

**Table 4-5 (Cont.) Oracle Reserved Namespaces**

| Namespace | Library |
|---|---|
| `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 |

## 4.4.8 Polling Mode Operations in OCI

OCI has calls that poll for completion.

Examples of such polling mode calls are:

- OCI calls in nonblocking mode
- OCI calls that operate on LOB data in pieces such as `OCILobRead2()` and `OCILobWrite2()`
- `OCIStmtExecute()` and `OCIStmtFetch2()` when used with `OCIStmtSetPieceInfo()` and `OCIStmtGetPieceInfo()`

In such cases, OCI requires that the application ensure that the same OCI call is repeated on the connection and nothing else is done on the connection in the interim. Performing any other OCI call on such a connection (when OCI has handed control back to the caller) can result in unexpected behavior.

Hence, with such polling mode OCI calls, the caller must ensure that the same call is repeated on the connection and that nothing else is done until the call completes.

`OCIBreak()` and `OCIReset()` are exceptions to the rule. These calls are allowed so that the caller can stop an OCI call that has been started.

## 4.4.9 Nonblocking Mode in OCI

OCI provides the ability to establish a server connection in *blocking mode* or *nonblocking mode*.

> **Note:**
>
> Because nonblocking mode requires the caller to repeat the same call until it completes, it increases CPU usage. Instead, use multithreaded mode.

> **See Also:**
>
> - Overview of OCI Multithreaded Development
> - OCIThread Package

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 must check whether the pending call is finished at the server by executing the call again *with the exact same parameters*.

The following features and functions are not supported in nonblocking mode:

- Direct Path Load
- LOB buffering
- Objects
- Query cache
- Scrollable cursors
- Transparent application failover (TAF)
- `OCIAQEnqArray()`
- `OCIAQDeqArray()`
- `OCIDescribeAny()`
- `OCILobArrayRead()`
- `OCILobArrayWrite()`
- `OCITransStart()`
- `OCITransDetach()`

## 4.4.10 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 `OCI_ATTR_NONBLOCKING_MODE` attribute only after `OCISessionBegin()` or `OCILogon2()` has been called. Otherwise, an error is returned.

**Note:**

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

> **See Also:**
>
> Server Handle Attributes

### 4.4.11 Canceling 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.

## 4.5 About 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 enables you to group some 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 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:**

- Although 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 must be initialized to `NULL` (through an indicator of -1, or an actual length of 0) before a PL/SQL begin-end block can be executed in OCI.
- OCI does not support the PL/SQL `RECORD` data type.
- When binding a PL/SQL VARCHAR2 variable in OCI, the maximum size of the bind variable is 32767 bytes, because of the overhead of control structures.

**Note:**

When you write PL/SQL code, it is important to remember that the parser treats everything between a pair of hyphens"--" and a carriage return character 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 ensure that the comment ends there.

**See Also:**

*Oracle Database PL/SQL Language Reference* for information about coding PL/SQL blocks

## 4.6 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.

This section includes the following topics:

- Client Character Set Control from OCI
- Character Control and OCI Interfaces
- Character-Length Semantics Support in Describe Operations
- Character Set Support in OCI

- Controlling Language and Territory in OCI
- Other OCI Globalization Support Functions
- About Getting Locale Information in OCI
- About OCI and the BOM (Byte Order Mark)
- About Manipulating Strings in OCI
- About Converting Character Sets in OCI
- OCI Messaging Functions
- lmsgen Utility

## 4.6.1 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.

`OCI_UTF16ID` cannot be set from `NLS_LANG` or `NLS_NCHAR` and must be set using `OCIEnvNlsCreate()`. 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. For example:

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

In this example, `csid` is the value for the character set ID for the parameter `charset`, and `ncsid` is the value for the national character set ID for the parameter `ncharset`. 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.

Any Oracle character set ID, except AL16UTF16, can be specified through the `OCIEnvNlsCtrate()` 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()
- About Setting Client Character Sets in OCI for a pseudocode fragment that illustrates a sample usage of these calls

## 4.6.2 Character Control and OCI Interfaces

How is character control performed by OCI interfaces.

The `OCINlsGetInfo()` function returns information about any character set, including `OCI_UTF16ID` if this value has been used in `OCIEnvNlsCreate()`.

The `OCIAttrGet()` function 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 0 by `OCIEnvNlsCreate()`, the character set IDs in `NLS_LANG` and `NLS_NCHAR` are returned.

The `OCIAttrSet()` function 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()` is passed as `charset` or `ncharset`.

The `OCIBindByName()` or `OCIBindByName2()` and `OCIBindByPos()` or `OCIBindByPos2()` functions bind variables with the 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.

The `OCIDefineByPos()` or `OCIDefineByPos2()` function 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.

## 4.6.3 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. (A single-byte character set is 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 the number of bytes is equal to a multiple of the number of characters. Therefore, scanning of the entire string is not needed to determine the number of characters for fixed-width character sets. However, for variable-width character sets, complete scanning is needed to determine the number of characters in a string.

## 4.6.4 Character Set Support in OCI

How does OCI support character sets.

See Character-Length Semantics Support in Describe Operations and Character Conversion in OCI Binding and Defining for a complete discussion of character set support in OCI.

## 4.6.5 Controlling Language and Territory in OCI

The NLS language and territory can also be set programmatically using the attributes `OCI_ATTR_ENV_NLS_LANGUAGE` and `OCI_ATTR_ENV_NLS_TERRITORY` on OCI environment handle.

See the following attributes for more details on their usage. These attributes will be effective for the database sessions created from that environment handle after the attributes have been set.

- OCI_ATTR_ENV_NLS_LANGUAGE
- OCI_ATTR_ENV_NLS_TERRITORY

## 4.6.6 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.

**See Also:**

- OCI Globalization Support Functions
- *Oracle Database Globalization Support Guide* for information about OCI programming with Unicode

## 4.6.7 About Getting Locale Information in OCI

An Oracle Database 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
- OCINlsEnvironmentVariableGet()

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

The code in Example 4-13 retrieves locale information and checks for errors.

**Example 4-13 Getting Locale Information in OCI**

```
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);
}
```

## 4.6.8 About OCI and the BOM (Byte Order Mark)

OCI does not support nor handle the BOM (byte order mark) and assumes that the byte order is native to the machine on which your application is executing.

Your OCI application must not pass a string containing a BOM expecting that OCI can detect the encoding of the string. Your OCI application must remove the BOM if it exists in the string being passed in and ensure that it is in the encoding the OCI function expects.

## 4.6.9 About 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 data type, `OCIWchar`, is Oracle-specific and should not be confused with the `wchar_t` data type defined by the ANSI/ISO C standard. The Oracle wide-character data type is always 4 bytes in all operating systems, whereas the size of `wchar_t` depends on the implementation and the operating system. The Oracle wide-character data type 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

Example 4-14 shows a simple case of manipulating strings.

The OCI character classification functions are described in detail in OCI Character Classification Functions.

Example 4-15 shows how to classify characters in OCI.

**Example 4-14 Basic String Manipulation in OCI**

```
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 4-15 Classifying 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);
}
```

## 4.6.10 About 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 an error is raised.

Example 4-16 shows a simple conversion into Unicode.

**See Also:**

OCI Character Set Conversion Functions

### Example 4-16    Converting Character Sets in OCI

```
/* 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);
}
```

## 4.6.11 OCI Messaging Functions

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

Example 4-17 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, and the product, facility, and cache size have been initialized properly.

**Example 4-17    Retrieving a Message from a Text Message File**

```
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);
```

**See Also:**

- *Oracle Database Data Cartridge Developer's Guide*
- OCI Messaging Functions

## 4.6.12 lmsgen Utility

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

The BNF syntax of the lmsgen utility is as follows:

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

In the preceding syntax:

- *`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.

This section includes the following topics:

- Guidelines for Text Message Files
- An Example of Creating a Binary Message File from a Text Message File

### 4.6.12.1 Guidelines for Text Message Files

What are the guidelines that text message files must follow.

Text message files must follow these guidelines:

- 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 three 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 Database 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)"
```

## 4.6.12.2 An Example of Creating a Binary Message File from a Text Message File

How do you create 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
```

## 5

# Data Types

This chapter provides a reference to Oracle external data types used by OCI applications.

It also discusses Oracle data types and the conversions between internal and external representations that occur when you transfer data between your program and an Oracle database.

This chapter contains these topics:

- Oracle Data Types
- Internal Data Types
- External Data Types
- Data Conversions
- Typecodes
- Definitions in oratypes.h

> **See Also:**
>
> *Oracle Database SQL Language Reference* for detailed information about Oracle internal data types

## 5.1 Oracle Data Types

One of the main functions of an OCI program is to communicate with an Oracle database.

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 data types*. Examples of internal data types include `NUMBER`, `CHAR`, and `DATE` (see Table 5-1).

In general, OCI applications do not work with internal data type representations of data, but with host language data types that 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 data types and *external data types*.

External data types 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 data type 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 data types 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 data types. They are discussed here because it can be useful to understand how internal types can map to external types.

OCI can perform a wide range of data type conversions when transferring data between an Oracle database and an OCI application. There are more OCI external data types than Oracle internal data types. In some cases, a single external type maps to an internal type; in other cases, multiple external types map to a single internal type.

The many-to-one mappings for some data types provide flexibility for the OCI programmer. For example, suppose that you are processing the following SQL statement:

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

You want the salary to be returned as character data, instead of a binary floating-point format. Therefore, you specify an Oracle database external string data type, such as `VARCHAR2` (code = 1) or `CHAR` (code = 96) for the `dty` parameter in the `OCIDefineByPos()` or `OCIDefineByPos2()` call for the `sal` column. You also must declare a string variable in your program and specify its address in the `valuep` parameter. See Table 5-2 for more information.

If you want the salary information to be returned as a binary floating-point value, however, specify the `FLOAT` (code = 4) external data type. You also must define a variable of the appropriate type for the `valuep` parameter.

Oracle Database performs most data conversions transparently. The ability to specify almost any external data type 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 data type.

To control data conversion, you must use the appropriate external data type codes in the bind and define routines. You must tell Oracle Database where the input or output variables are in your OCI program and their data types and lengths.

OCI also supports an additional set of OCI typecodes that are used by the Oracle Database type management system to represent data types of object type attributes. You can use a set of predefined constants to represent these typecodes. The constants each contain the prefix `OCI_TYPECODE`.

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

- Internal Oracle data types, which are used by table columns in an Oracle database. These also include data types used by PL/SQL that are not used by Oracle Database columns (for example, indexed table, boolean, record).
- External OCI data types, which are used to specify host language representations of Oracle data.
- `OCI_TYPECODE` values, which are used by Oracle Database to represent type information for object type attributes.

Information about a column's internal data type is conveyed to your application in the form of an internal data type code. With this information about what type of data is to be returned, your application can determine how to convert and format the output data. The Oracle internal data type codes are listed in the section Internal Data Types.

> **See Also:**
>
> - DATE for a description of the external data type
> - Internal Data Types
> - External Data Types and About Using External Data Type Codes
> - Typecodes, and Relationship Between SQLT and OCI_TYPECODE Values
> - *Oracle Database SQL Language Reference* for detailed information about Oracle internal data types
> - About Describing Select-List Items for information about describing select-list items in a query

## 5.1.1 About Using External Data Type Codes

An external data type code indicates to Oracle Database how a host variable represents data in your program.

This determines how the data is converted when it is returned to output variables in your program, or how it is converted from input (bind) variables to Oracle Database column values. For example, to convert a `NUMBER` in an Oracle database column to a variable-length character array, you specify the `VARCHAR2` external data type code in the `OCIDefineByPos()` call that defines the output variable.

To convert a bind variable to a value in an Oracle Database column, specify the external data type code that corresponds to the type of the bind variable. For example, to input a character string such as 02-FEB-65 to a `DATE` column, specify the data type as a character string and set the length parameter to 9.

It is always the programmer's responsibility to ensure that values are convertible. If you try to insert the string `"MY BIRTHDAY"` into a `DATE` column, you get an error when you execute the statement.

**See Also:**

Table 5-2 for a complete list of the external data types and data type codes

# 5.2 Internal Data Types

Lists and describes the internal data types.

Table 5-1 lists the internal Oracle Database data types (also known as *built-in*), along with each type's maximum internal length and data type code. PL/SQL types listed in Table 5-10 and Table 5-11 are also considered to be internal data types.

**Table 5-1 Internal Oracle Database Data Types**

| Internal Oracle Database Data Type | Maximum Internal Length | Data Type Code |
|---|---|---|
| `VARCHAR2, NVARCHAR2` | 4000 bytes (standard)<br>32767 bytes (extended) | 1 |
| `NUMBER` | 21 bytes | 2 |
| `LONG` | 2^31-1 bytes (2 gigabytes) | 8 |
| `DATE` | 7 bytes | 12 |
| `RAW` | 2000 bytes (standard)<br>32767 bytes (extended) | 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`) | Not Applicable | 108 |
| `REF` | Not Applicable | 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 |

This section includes the following topics:

- LONG, RAW, LONG RAW, VARCHAR2
- Character Strings and Byte Arrays
- UROWID
- BINARY_FLOAT and BINARY_DOUBLE

> **See Also:**
>
> *Oracle Database SQL Language Reference* for more information about these built-in data types

## 5.2.1 LONG, RAW, LONG RAW, VARCHAR2

Use piecewise capabilities provided by specific OCI APIs to perform inserts, updates or fetches of these data types.

You can use the piecewise capabilities provided by `OCIBindByName()` or `OCIBindByName2()`, `OCIBindByPos()` or `OCIBindByPos2()`, `OCIDefineByPos()` or `OCIDefineByPos2()`, `OCIStmtGetPieceInfo()`, and `OCIStmtSetPieceInfo()` to perform inserts, updates or fetches involving column data of the `LONG`, `RAW`, `LONG RAW`, and `VARCHAR2` data types.

> **See Also:**
>
> - OCIBindByName() or OCIBindByName2()
> - OCIBindByPos() or OCIBindByPos2()
> - OCIDefineByPos() or OCIDefineByPos2()
> - OCIStmtGetPieceInfo()
> - OCIStmtSetPieceInfo()

## 5.2.2 Character Strings and Byte Arrays

Use Oracle internal data types to specify columns that contain characters or arrays of bytes.

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

> **Note:**
>
> LOBs can contain characters and `BFILE`s can contain binary data. They are handled differently than other types, so they are not included in this discussion.

`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 it is 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 data type 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 Database 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.

> **See Also:**
>
> LOB and BFILE Operations for more information about `CHAR`, `VARCHAR2`, `RAW`, `LONG`, and `LONG RAW` data types

## 5.2.3 UROWID

The Universal `ROWID` (`UROWID`) is a data type that can store both logical and physical rowids of Oracle Database tables.

Logical rowids are primary key-based logical identifiers for the rows of index-organized tables (IOTs).

To use columns of the `UROWID` data type, the value of the `COMPATIBLE` initialization parameter must be set to 8.1 or later.

The following host variables can be bound to Universal `ROWID`s:

- `SQLT_CHR` (`VARCHAR2`)
- `SQLT_VCS` (`VARCHAR`)
- `SQLT_STR` (`NULL`-terminated string)
- `SQLT_LVC` (`LONG VARCHAR`)
- `SQLT_AFC` (`CHAR`)
- `SQLT_AVC` (`CHARZ`)
- `SQLT_VST` (OCI String)
- `SQLT_RDD` (`ROWID` descriptor)

## 5.2.4 BINARY_FLOAT and BINARY_DOUBLE

The `BINARY_FLOAT` and `BINARY_DOUBLE` data types 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 data types with release 10.1, all numeric values in an Oracle Database were stored in the Oracle `NUMBER` format. These new binary floating point types do 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` data types:

- `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, use external types `SQLT_BFLOAT` and `SQLT_BDOUBLE` in conjunction with the `BINARY_FLOAT` and `BINARY_DOUBLE` data types.

# 5.3 External Data Types

Lists and describes the data type codes for external data types.

Table 5-2 lists data type codes for external data types. For each data type, the table lists the program variable types for C from or to which Oracle Database internal data is normally converted.

### Table 5-2 External Data Types and Codes

| External Data Type | Code | Program Variable[1] | 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` |
| 64-bit signed `INTEGER` | 3 | signed long, signed long 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` |
| `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[2] | `SQLT_CLOB` |
| Binary LOB descriptor | 113 | OCILobLocator[2] | `SQLT_BLOB` |
| Binary FILE descriptor | 114 | OCILobLocator | `SQLT_FILE` |
| `OCI STRING` type | 155 | OCIString | `SQLT_VST`[3] |
| `OCI DATE` type | 156 | OCIDate * | `SQLT_ODT`[3] |
| `ANSI DATE` descriptor | 184 | OCIDateTime * | `SQLT_DATE` |
| `TIMESTAMP` descriptor | 187 | OCIDateTime * | `SQLT_TIMESTAMP` |
| `TIMESTAMP WITH TIME ZONE` descriptor | 188 | OCIDateTime * | `SQLT_TIMESTAMP_TZ` |

**Table 5-2 (Cont.) External Data Types and Codes**

| External Data Type | Code | Program Variable[1] | OCI-Defined Constant |
|---|---|---|---|
| `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` |

[1] Where the length is shown as n, it is a variable, and depends on the requirements of the program (or of the operating system for ROWID).

[2] In applications using data type mappings generated by OTT, CLOBs may be mapped as OCIClobLocator, and BLOBs may be mapped as OCIBlobLocator. For more information, see Chapter 15.

[3] For more information about the use of these data types, see Chapter 12.

This section includes the following topics describing these external data types:

- VARCHAR2
- NUMBER
- 64-Bit Integer Host Data Type
- INTEGER
- FLOAT
- STRING
- VARNUM
- LONG
- VARCHAR
- DATE
- RAW
- VARRAW
- LONG RAW
- UNSIGNED
- LONG VARCHAR
- LONG VARRAW
- CHAR
- CHARZ
- Named Data Types: Object, VARRAY, Nested Table
- REF
- ROWID Descriptor
- LOB Descriptor
- Datetime and Interval Data Type Descriptors
- Native Float and Native Double

- C Object-Relational Data Type Mappings

## 5.3.1 VARCHAR2

The `VARCHAR2` data type is a variable-length string of characters with a maximum length of 4000 bytes.

If the `init.ora` parameter `max_string_size = standard` (default value), the maximum length of a `VARCHAR2` can be 4000 bytes. If the `init.ora` parameter `max_string_size = extended`, the maximum length of a `VARCHAR2` can be 32767 bytes.

> **Note:**
>
> If you are using Oracle Database objects, you can work with a special `OCIString` external data type using a set of predefined OCI functions.

This section includes the following topics:

- Input
- Output

> **See Also:**
>
> - `init.ora` parameter MAX_STRING_SIZE in *Oracle Database Reference* for more information about extended data types
> - Object-Relational Data Types in OCI for more information about the `OCIString` external data type

### 5.3.1.1 Input

The `value_sz` parameter determines the length in the `OCIBindByName()` or `OCIBindByPos()` call. If you are using extended `VARCHAR2` lengths, then the `value_sz` parameter determines the length in the `OCIBindByName2()` and `OCIBindByPos2()` calls.

If the `value_sz` parameter is greater than zero, Oracle Database 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, with 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 Database 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 Database issues an error, and the row is not inserted.

When the Oracle internal (column) data type 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 Database returns an error and the value is not inserted into the database.

**See Also:**

- OCIBindByName()
- OCIBindByPos()
- OCIBindByName2()
- OCIBindByPos2()

## 5.3.1.2 Output

You must specify the desired length for the return value in `value_sz` for bind and define functions.

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 are using extended `VARCHAR2` lengths, then the `value_sz` parameter determines the desired length for the return value in the `OCIDefineByPos2()` call, or in the `OCIBindByName2()` and `OCIBindByPos2()` calls for PL/SQL blocks.

If you omit the `rlenp` parameter of `OCIDefineByPos()`, returned values are blank-padded to the buffer length, and `NULL`s 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 Database 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 for the error returns `ORA-1405`.

**See Also:**

- Indicator Variables
- OCIDefineByPos() or OCIDefineByPos2()
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()

## 5.3.2 NUMBER

You should not need to use `NUMBER` as an external data type.

If you do use it as an external data type, Oracle Database returns numeric values in its internal 21-byte binary format and expects this format on input. The following discussion is included for completeness only.

**Note:**

If you are using objects in an Oracle database, you can work with a special `OCINumber` data type using a set of predefined OCI functions.

Oracle Database stores values of the `NUMBER` data type 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 zeros 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 data type code 2 in the `dty` parameter of an `OCIDefineByPos()` or `OCIDefineByPos2()` 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 must know the number of bytes returned, use the `VARNUM` external data type instead of `NUMBER`.

**See Also:**

- OCINumber Examples
- VARNUM for a description of the internal `NUMBER` format
- Number (OCINumber) more information about the `OCINumber` data type
- OCIDefineByPos() or OCIDefineByPos2()

## 5.3.3 64-Bit Integer Host Data Type

You can bind and define integer values greater than 32-bit size (more than nine digits of precision) from and into a `NUMBER` column using a 64-bit native host variable and `SQLT_INT` or `SQLT_UIN` as the external data type in an OCI application.

Starting with release 11.2, OCI supports the ability to bind and define integer values greater than 32-bit size (more than nine digits of precision) from and into a `NUMBER` column using a 64-bit native host variable and `SQLT_INT` or `SQLT_UIN` as the external data type in an OCI application.

This feature enables an application to bind and define 8-byte native host variables using `SQLT_INT` or `SQLT_UIN` external data types in the OCI bind and define function calls on all platforms. The `OCIDefineByPos()` or `OCIDefineByPos2()`, `OCIBindByName()` or `OCIBindByName2()`, and `OCIBindByPos()` or `OCIBindByPos2()` function calls can specify an 8-byte integer data type pointer as the `valuep` parameter. This feature enables you to insert and fetch large integer values (up to 18 decimal digits of precision) directly into and from native host variables and to perform free arithmetic on them.

This section includes the following topics:

- OCI Bind and Define for 64-Bit Integers
- Support for OUT Bind DML Returning Statements
- OCIDefineByPos() or OCIDefineByPos2()
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()

### 5.3.3.1 OCI Bind and Define for 64-Bit Integers

Shows a code fragment for an OCI bind and define for 64-bit integers.

Example 5-1 shows a code fragment that works without errors.

**Example 5-1 OCI Bind and Define Support for 64-Bit Integers**

```
...
/* Variable declarations */
orasb8    sbigval1, sbigval2, sbigval3; // Signed 8-byte variables.
oraub8    ubigval1, ubigval2, ubigval3; // Unsigned 8-byte variables.
```

```
...
/* Bind Statements */
OCIBindByPos(..., (void *) &sbigval1, sizeof(sbigval1), ..., SQLT_INT, ...);
OCIBindByPos(..., (void *) &ubigval1, sizeof(ubigval1), ..., SQLT_UIN, ...);
OCIBindByName(...,(void *) &sbigval2, sizeof(sbigval2), ..., SQLT_INT, ...);
OCIBindByName(...,(void *) &ubigval2, sizeof(ubigval2), ..., SQLT_UIN, ...);
...
/* Define Statements */
OCIDefineByPos(..., (void *) &sbigval3, sizeof(sbigval3), ..., SQLT_INT, ...);
OCIDefineByPos(..., (void *) &ubigval3, sizeof(ubigval3), ..., SQLT_UIN, ...);
...
```

### 5.3.3.2 Support for OUT Bind DML Returning Statements

Shows a code fragment that illustrates binding 8-byte integer data types for OUT binds of a DML returning statement.

Example 5-2 shows a code fragment that illustrates binding 8-byte integer data types for OUT binds of a DML returning statement.

**Example 5-2 Binding 8-Byte Integer Data Types for OUT Binds of a DML Returning Statement**

```
...
/* Define SQL statements to be used in program. */
static text *dml_stmt = (text *) " UPDATE emp SET sal = sal + :1
                                   WHERE empno = :2
                                   RETURNING sal INTO :out1";
...

/* Declare all handles to be used in program. */
OCIStmt    *stmthp;
OCIError   *errhp;
OCIBind    *bnd1p   = (OCIBind *) 0;
OCIBind    *bnd2p   = (OCIBind *) 0;
OCIBind    *bnd3p   = (OCIBind *) 0;
...

/* Bind variable declarations */
orasb8    sbigval;   // OUT bind variable (8-byte size).
sword     eno, hike; // IN bind variables.
...

/* get values for IN bind variables */
...

/* Bind Statements */
OCIBindByPos(stmthp, &bnd1p, errhp, 1, (dvoid *) &hike,
            (sb4) sizeof(hike), SQLT_INT, (dvoid *) 0,
            (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT);
OCIBindByPos(stmthp, &bnd2p, errhp, 2, (dvoid *) &eno,
            (sb4) sizeof(eno), SQLT_INT, (dvoid *) 0,
            (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT);
OCIBindByName(stmthp, &bnd3p, errhp, (text *) ":out1", -1,
             (dvoid *) &sbigval, sizeof(sbigval), SQLT_INT, (dvoid *) 0,
             (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT);
...

/* Use the returned OUT bind variable value */
...
```

## 5.3.4 INTEGER

The `INTEGER` data type 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 Database 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 Database returns an "overflow on conversion" error.

## 5.3.5 FLOAT

The `FLOAT` data type 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 4 or 8 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 Database 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.

## 5.3.6 STRING

The `NULL`-terminated `STRING` format behaves like the `VARCHAR2` format, except that the string must contain a `NULL` terminator character.

The `STRING` data type is most useful for C language programs.

This section includes the following topics:

- Input
- Output

### 5.3.6.1 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 Database issues the following error:

ORA-01480: trailing `NULL` missing from `STR` bind value

If the length is not specified in the bind call, OCI uses an implied maximum string length of 4000.

The minimum string length is 2 bytes. If the first character is a `NULL` terminator and the length is specified as 2, a `NULL` is inserted into 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:**
>
> You cannot pass -1 for the string length parameter of a `NULL`-terminated string

### 5.3.6.2 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 also possible.

## 5.3.7 VARNUM

The `VARNUM` data type is like the external `NUMBER` data type, 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 5-3 shows several examples of the `VARNUM` values returned for numbers in a table.

**Table 5-3 VARNUM Examples**

| Decimal Value | Length Byte | Exponent Byte | Mantissa Bytes | Terminator Byte |
|---|---|---|---|---|
| 0 | 1 | 128 | Not applicable | Not applicable |
| 5 | 2 | 193 | 6 | Not applicable |
| -5 | 3 | 62 | 96 | 102 |
| 2767 | 3 | 194 | 28, 68 | Not applicable |
| -2767 | 4 | 61 | 74, 34 | 102 |
| 100000 | 2 | 195 | 11 | Not applicable |
| 1234567 | 5 | 196 | 2, 24, 46, 68 | Not applicable |

## 5.3.8 LONG

The `LONG` data type stores character strings longer than 4000 bytes.

You can store up to 2 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.

## 5.3.9 VARCHAR

The `VARCHAR` data type stores character strings of varying length.

The first 2 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.

## 5.3.10 DATE

The `DATE` data type can update, insert, or retrieve a date value using the Oracle internal date binary format.

A date in binary format contains 7 bytes, as shown in Table 5-4.

**Table 5-4 Format of the DATE Data Type**

| 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 data type, 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` data type 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.

If you are using objects in an Oracle database, you can work with a special `OCIDate` data type using a set of predefined OCI functions.

**See Also:**

- Date (OCIDate) for more information about the `OCIDate` data type
- Datetime and Interval Data Type Descriptors for information about `DATETIME` and `INTERVAL` data types

## 5.3.11 RAW

The `RAW` data type is used for binary data or byte strings that are not to be interpreted by Oracle Database, for example, to store graphics character sequences.

The maximum length of a `RAW` column is 2000 bytes. If the `init.ora` parameter `max_string_size = standard` (default value), the maximum length of a `RAW` can be 2000 bytes. If the `init.ora` parameter `max_string_size = extended`, the maximum length of a `RAW` can be 32767 bytes.

When `RAW` data in an Oracle Database 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'. To input a character string in your program to a `RAW` column in an Oracle Database 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.

If you are using objects in an Oracle database, you can work with a special `OCIRaw` data type using a set of predefined OCI functions.

**See Also:**

- *Oracle Database SQL Language Reference* for more information about `MAX_STRING_SIZE`
- `init.ora` parameter MAX_STRING_SIZE in *Oracle Database Reference* for more information about extended data types
- Raw (OCIRaw)for more information about this data type

## 5.3.12 VARRAW

The `VARRAW` data type is similar to the `RAW` data type.

However, the first 2 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 data type.

## 5.3.13 LONG RAW

The `LONG RAW` data type supports a 2 gigabyte length.

The `LONG RAW` data type is similar to the `RAW` data type, except that it stores raw data with a length up to 2 gigabytes (2^31-1 bytes).

## 5.3.14 UNSIGNED

The `UNSIGNED` data type 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 Database 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 Database returns an "overflow on conversion" error.

## 5.3.15 LONG VARCHAR

The `LONG VARCHAR` data type stores data from and into an Oracle Database `LONG` column.

The first 4 bytes of a `LONG VARCHAR` contain the length of the item. So, the maximum length of a stored item is 2^31-5 bytes.

## 5.3.16 LONG VARRAW

The `LONG VARRAW` data type is used to store data from and into an Oracle Database `LONG RAW` column.

The length is contained in the first four bytes. The maximum length is 2^31-5 bytes.

## 5.3.17 CHAR

The `CHAR` data type is a string of characters, with a maximum length of 2000.

`CHAR` strings are compared using blank-padded comparison semantics.

This section includes the following topics:

- Input
- Output

> **See Also:**
>
> *Oracle Database SQL Language Reference*

### 5.3.17.1 Input

The length is determined by the `value_sz` parameter in the `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`call.

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

If the `value_sz` parameter is zero, Oracle Database 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 Database issues an error and does not insert the row.

Negative values for the `value_sz` parameter are not allowed for `CHAR`s.

When the Oracle internal (column) data type 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 Database 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.

> **See Also:**
>
> - OCIBindByName() or OCIBindByName2()
> - OCIBindByPos() or OCIBindByPos2()

### 5.3.17.2 Output

Specify the desired length for the return value in the `value_sz` parameter of the `OCIDefineByPos()` or `OCIDefineByPos2()` call.

If zero is specified for the length, no data is returned.

If you omit the `rlenp` parameter of `OCIDefineByPos()` or `OCIDefineByPos2()`, returned values are blank padded to the buffer length, and `NULL`s 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 character truncation occurs, include an indicator parameter or array of indicator parameters in the `OCIDefineByPos()` or `OCIDefineByPos2()` 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.

You can also request output to a character string from an internal `NUMBER` data type. 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.

**See Also:**

- Indicator Variables
- OCIDefineByPos() or OCIDefineByPos2()

## 5.3.18 CHARZ

The `CHARZ` external data type is similar to the `CHAR` data type, except that the string must be `NULL`-terminated on input, and Oracle Database 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 follows, then the length parameter when you bind `my_num` must be seven. Any other value would return an error for this example.

```
char my_num[] = "123.45";
```

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

**Note:**

Both internal and external data types have Oracle-defined constant values, such as `SQLT_NTY`, `SQLT_REF`, corresponding to their data type 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 data types. The data type constants are also used in other chapters of this guide when referring to these new types.

## 5.3.19 Named Data Types: Object, VARRAY, Nested Table

Named data types are user-defined types that are specified with the `CREATE TYPE` command in SQL.

Examples include object types, varrays, and nested tables. In OCI, *named data type* refers to a host language representation of the type. The `SQLT_NTY` data type code is used when binding or defining named data types.

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

**See Also:**

- Object Type Information Storage and Access for more information about working with named data types in OCI
- Using the Object Type Translator with OCI for information about how named data types are represented as C structs

## 5.3.20 REF

This is a reference to a named data type.

The C language representation of a `REF` is a variable declared to be of type `OCIRef *`. The `SQLT_REF` data type code is used when binding or defining `REF`s.

Access to `REF`s is only possible when an OCI application has been initialized in object mode. When `REF`s 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:**

- OCIObjectNew()
- Object Advanced Topics in OCI for more information about working with `REF`s in the OCI

## 5.3.21 ROWID Descriptor

The `ROWID` data type 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 by using `OCIAttrGet()` on the statement handle and used in a subsequent `UPDATE` statement. The prefetch operation fetches all `ROWID`s on a `SELECT` for `UPDATE`; use prefetching and then a single row fetch.

You access rowids using a `ROWID` descriptor, which you can use as a bind or define variable.

**See Also:**

- OCIAttrGet()
- OCI Descriptors and Positioned Updates and Deletes for more information about the use of the `ROWID` descriptor

## 5.3.22 LOB Descriptor

A LOB (large object) stores binary or character data up to 128 terabytes (TB) 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 Oracle database. 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 that includes a LOB column or attribute in its select list, fetching the results 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.

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.

The data type codes available for binding or defining LOBs are:

- `SQLT_BLOB` - A binary LOB data type
- `SQLT_CLOB` - A character LOB data type

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

- To write into or read from an `NCLOB`, the user must set the character set form (`csfrm`) parameter to be `SQLCS_NCHAR`.
- The amount (`amtp`) parameter in calls involving `CLOB`s and `NCLOB`s is always interpreted in terms of characters, rather than bytes, for fixed-width character sets.

This section includes the following topics:

- BFILE
- BLOB
- CLOB
- NCLOB

**See Also:**

- OCI Descriptors for more information about descriptors, including the LOB locator
- *Oracle Database SQL Language Reference* and *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about LOBs
- About Binding LOB Data
- About Defining LOB Data
- LOB and BFILE Functions in OCI
- OCIDescriptorAlloc()
- OCIObjectPin()
- LOB and BFILE Operations for more information about OCI LOB functions

### 5.3.22.1 BFILE

Oracle Database supports access to binary files (`BFILE`s).

The `BFILE` data type 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 file name. 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 Database processes have operating system read permissions on the file.

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

The data type code available for binding or defining `BFILE`s is `SQLT_BFILE` (a binary FILE LOB data type)

**See Also:**

*Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about directory aliases

### 5.3.22.2 BLOB

The `BLOB` data type stores unstructured binary large objects.

`BLOB`s can be thought of as bit streams with no character set semantics. `BLOB`s can store up to 128 terabytes of binary data.

`BLOB`s have full transactional support; changes made through 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.

### 5.3.22.3 CLOB

The `CLOB` data type stores fixed-width or variable-width character data.

`CLOB`s can store up to 128 terabytes of character data.

`CLOB`s have full transactional support; changes made through 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.

### 5.3.22.4 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. `NCLOB`s can store up to 128 terabytes of character text data.

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

## 5.3.23 Datetime and Interval Data Type Descriptors

Lists and describes the datetime and interval data type descriptors.

The datetime and interval data type descriptors are briefly summarized here.

This section includes the following topics:

- ANSI DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND
- About Avoiding Unexpected Results Using Datetime

> **See Also:**
>
> *Oracle Database SQL Language Reference*

### 5.3.23.1 ANSI DATE

`ANSI DATE` is based on `DATE`, but contains no time portion. It also has no time zone.

`ANSI DATE` follows the ANSI specification for the `DATE` data type. When assigning an `ANSI DATE` to a `DATE` or a time stamp data type, the time portion of the Oracle `DATE` and the time stamp are set to zero. When assigning a `DATE` or a time stamp to an `ANSI DATE`, the time portion is ignored.

Instead of using the `ANSI DATE` data type, Oracle recommends that you use the `TIMESTAMP` data type, which contains both date and time.

### 5.3.23.2 TIMESTAMP

The `TIMESTAMP` data type is an extension of the `DATE` data type. It stores the year, month, and day of the `DATE` data type, plus the hour, minute, and second values.

The `TIMESTAMP` data type has no time zone. The `TIMESTAMP` data type has the following form:

```
TIMESTAMP(fractional_seconds_precision)
```

In this form, 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.

### 5.3.23.3 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` data type has the following form:

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

In this form, `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.

### 5.3.23.4 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 Database 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` data type has the following form:

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

In this form, `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.

### 5.3.23.5 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` data type has the following form:

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

In this form, the optional `year_precision` is the number of digits in the `YEAR` datetime field. The default value of `year_precision` is 2.

### 5.3.23.6 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` data type has the following form:

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

In this form:

- `day_precision` is the number of digits in the `DAY` datetime field. It is optional. Accepted values are 0 to 9. The default is 2.
- `fractional_seconds_precision` is the number of digits in the fractional part of the `SECOND` datetime field. Accepted values are 0 to 9. The value should be provided as nanoseconds. The Default Day to Second precision is 6 unless the precision is specified to a different value at the time of creating the table. In this case, the least significant three digits will be truncated.

### 5.3.23.7 About Avoiding Unexpected Results Using Datetime

How to avoid unexpected results using datetime.

**Note:**

To avoid unexpected results in your data manipulation language (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 Database uses the operating system time zone by default. If the operating system time zone is not a valid Oracle Database time zone, Oracle Database uses UTC as the default value.

### 5.3.24 Native Float and Native Double

The native float (`SQLT_BFLOAT`) and native double (`SQLT_BDOUBLE`) data types represent the single-precision and double-precision floating-point values.

They are represented natively, that is, in the host system's floating-point format.

These external types were added in release 10.1 to externally represent the `BINARY_FLOAT` and `BINARY_DOUBLE` internal data types. Thus, performance for the internal types is 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 types.

### 5.3.25 C Object-Relational Data Type Mappings

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

OCI provides a set of calls to operate on these data types, and to use these data types in bind and define operations, in conjunction with OCI external data types.

**See Also:**

Object-Relational Data Types in OCI for information about using these Oracle-defined C data types

## 5.4 Data Conversions

Shows the supported conversions from internal data types to external data types and from external data types into internal column representations.

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

- REFs stored in the database are converted to `SQLT_REF` on output.
- `SQLT_REF` is converted to the internal representation of `REF`s on input.

- Named data types stored in the database can be converted to `SQLT_NTY` (and represented by a C struct in the application) on output.
- `SQLT_NTY` (represented by a C struct in an application) is converted to the internal representation of the corresponding type on input.

LOBs are shown in Table 5-6, because of the width limitation.

> **See Also:**
>
> Object-Relational Data Types in OCI for information about `OCIString`, `OCINumber`, and other new data types

**Table 5-5 Data Conversions**

| **NA[1]** | **INTERNAL DATA TYPES->** | **NA** | **NA** | **NA** | **NA** | **NA** | **NA** | **NA** | **NA** |
|---|---|---|---|---|---|---|---|---|---|
| **EXTERNAL DATA TYPES** | **VARCHAR2** | **NUMBER** | **LONG** | **ROWID** | **UROWID** | **DATE** | **RAW** | **LONG RAW** | **CHAR** |
| `VARCHAR2` | I/O[2] | I/O | I/O | I/O[3] | I/O[3] | I/O[4] | I/O[5] | I/O[5] | NA |
| `NUMBER` | I/O[6] | I/O | I[7] | NA | NA | NA | NA | NA | I/O[6] |
| `INTEGER` | I/O[6] | I/O | I | NA | NA | NA | NA | NA | I/O[6] |
| `FLOAT` | I/O[6] | I/O | I | NA | NA | NA | NA | NA | I/O[6] |
| `STRING` | I/O | I/O | I/O | I/O[3] | I/O[3] | I/O[4] | I/O[5] | I/O[5,8] | I/O |
| `VARNUM` | I/O[6] | I/O | I | NA | NA | NA | NA | NA | I/O[6] |
| `DECIMAL` | I/O[6] | I/O | I | NA | NA | NA | NA | NA | I/O[6] |
| `LONG` | I/O | I/O | I/O | I/O[3] | I/O[3] | I/O[4] | I/O[5] | I/O[5,8] | I/O |
| `VARCHAR` | I/O | I/O | I/O | I/O[3] | I/O[3] | I/O[4] | I/O[5] | I/O[5,8] | I/O |
| `DATE` | I/O | NA | I | NA | NA | I/O | NA | NA | I/O |
| `VARRAW` | I/O[9] | NA | I[8,9] | NA | NA | NA | I/O | I/O | I/O[9] |
| `RAW` | I/O[9] | NA | I[8,9] | NA | NA | NA | I/O | I/O | I/O[9] |
| `LONG RAW` | O[10,9] | NA | I[8,9] | NA | NA | NA | I/O | I/O | O[9] |
| `UNSIGNED` | I/O[6] | I/O | I | NA | NA | NA | NA | NA | I/O[6] |
| `LONG VARCHAR` | I/O | I/O | I/O | I/O[3] | I/O[3] | I/O[4] | I/O[5] | I/O[5,8] | I/O |
| `LONG VARRAW` | I/O[9] | NA | I[8,9] | NA | NA | NA | I/O | I/O | I/O[9] |
| `CHAR` | I/O | I/O | I/O | I/O[3] | I/O[3] | I/O[4] | I/O[5] | I[5] | I/O |
| `CHARZ` | I/O | I/O | I/O | I/O[3] | I/O[3] | I/O[4] | I/O[5] | I[5] | I/O |
| `ROWID` descriptor | I[3] | NA | NA | I/O | I/O | NA | NA | NA | I[3] |

---

[1] NA means not applicable.

[2] I/O = Conversion is valid for input or output.

[3] For input, host string must be in Oracle ROWID/UROWID format. On output, column value is returned in Oracle ROWID/UROWID format.

[4] For input, host string must be in the Oracle DATE character format. On output, column value is returned in Oracle DATE format.

[5] For input, host string must be in hexadecimal format. On output, column value is returned in hexadecimal format.

[6] For output, column value must represent a valid number.

[7] I = Conversion is valid for input only.

[8] Length must be less than or equal to 2000.

[9] On input, column value is stored in hexadecimal format. On output, column value must be in hexadecimal format.

[10] O = Conversion is valid for output only.

This section includes the following topics:

- Data Conversions for LOB Data Type Descriptors
- Data Conversions for Datetime and Interval Data Types
- Datetime and Date Upgrading Rules
- Data Conversion for BINARY_FLOAT and BINARY_DOUBLE in OCI

## 5.4.1 Data Conversions for LOB Data Type Descriptors

Shows the data conversions for `LOB`s.

Table 5-6 shows the data conversions for `LOB`s. For example, the external character data types (`VARCHAR`, `CHAR`, `LONG`, and `LONG VARCHAR`) convert to the internal `CLOB` data type, whereas the external raw data types (`RAW`, `VARRAW`, `LONG RAW`, and `LONG VARRAW`) convert to the internal `BLOB` data type.

**Table 5-6    Data Conversions for LOBs**

| EXTERNAL DATA TYPES | INTERNAL CLOB | INTERNAL BLOB |
|---|---|---|
| `VARCHAR` | I/O[1] | NA[2] |
| `CHAR` | I/O | NA |
| `LONG` | I/O | NA |
| `LONG VARCHAR` | I/O | NA |
| `RAW` | NA | I/O |
| `VARRAW` | NA | I/O |
| `LONG RAW` | NA | I/O |
| `LONG VARRAW` | NA | I/O |

[1] I/O = Conversion is valid for input or output.

[2] NA means not applicable.

## 5.4.2 Data Conversions for Datetime and Interval Data Types

Shows the data conversion for datetime and interval data types.

You can also use one of the character data types for the host variable used in a fetch or insert operation from or to a datetime or interval column. Oracle Database does the conversion between the character data type and datetime or interval data type for you (see Table 5-7.

**Table 5-7 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[1] | I/O | I/O | I/O | I/O | I/O | I/O |
| `DATE` | I/O | I/O | I/O | I/O | I/O | NA[2] | NA |
| `OCI DATE` | I/O | I/O | I/O | I/O | I/O | NA | NA |
| `ANSI DATE` | I/O | I/O | I/O | I/O | I/O | NA | NA |
| `TIMESTAMP (TS)` | I/O | I/O | I/O | I/O | I/O | NA | NA |
| `TIMESTAMP WITH TIME ZONE (TSTZ)` | I/O | I/O | I/O | I/O | I/O | NA | NA |
| `TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)` | I/O | I/O | I/O | I/O | I/O | NA | NA |
| `INTERVAL YEAR TO MONTH` | I/O | NA | NA | NA | NA | I/O | NA |
| `INTERVAL DAY TO SECOND` | I/O | NA | NA | NA | NA | NA | I/O |

[1] I/O = Conversion is valid for input or output.

[2] NA means not applicable.

This section includes the following topics:

- Assignment Notes
- Data Conversion Notes for Datetime and Interval Types

## 5.4.2.1 Assignment Notes

When you assign a source with a time zone to a target without a time zone, the time zone portion of the source is ignored.

When you assign 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 you assign an Oracle Database `DATE` to a `TIMESTAMP`, the `TIME` portion of the `DATE` is copied over to the `TIMESTAMP`. When you assign a `TIMESTAMP` to Oracle Database `DATE`, the `TIME` portion of the result `DATE` is set to zero. This is done to encourage upgrading of Oracle Database `DATE` to `ANSI`-compliant `DATETIME` data types.

When you assign an `ANSI DATE` to an Oracle `DATE` or a `TIMESTAMP`, the `TIME` portion of the Oracle Database `DATE` and the `TIMESTAMP` are set to zero. When you assign an Oracle Database `DATE` or a `TIMESTAMP` to an ANSI `DATE`, the `TIME` portion is ignored.

When you assign a `DATETIME` to a character string, the `DATETIME` is converted using the session's default `DATETIME` format. When you assign a character string to a `DATETIME`, the string must contain a valid `DATETIME` value based on the session's default `DATETIME` format

When you assign a character string to an `INTERVAL`, the character string must be a valid `INTERVAL` character format.

### 5.4.2.2 Data Conversion Notes for Datetime and Interval Types

Describes some information for datetime and interval types.

When you convert from `TSLTZ` to `CHAR`, `DATE`, `TIMESTAMP`, and `TSTZ`, the value is adjusted to the session time zone.

When you convert from `CHAR`, `DATE`, and `TIMESTAMP` to `TSLTZ`, the session time zone is stored in memory.

When you assign `TSLTZ` to `ANSI DATE`, the time portion is zero.

When you convert from `TSTZ`, the time zone that the time stamp is in is stored in memory.

When you assign a character string to an interval, the character string must be a valid interval character format.

## 5.4.3 Datetime and Date Upgrading Rules

OCI has full forward and backward compatibility between a client application and the Oracle database for datetime and date columns.

This section includes the following topics:

- Pre-9.0 Client with 9.0 or Later Server
- Pre-9.0 Server with 9.0 or Later Client

### 5.4.3.1 Pre-9.0 Client with 9.0 or Later Server

The only datetime data type available to a pre-9.0 application is the `DATE` data type, `SQLT_DAT`.

When a pre-9.0 client that defined a buffer as `SQLT_DAT` tries to obtain data from a `TSLTZ` column, only the date portion of the value is returned to the client.

### 5.4.3.2 Pre-9.0 Server with 9.0 or Later Client

When a pre-9.0 server is used with a 9.0 or later client, the 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 data types) into a `DATE` column, an error is issued because there is potential data loss in this situation.

When a client has an OUT bind or a define buffer that is of data type `SQLT_TIMESTAMP_LTZ` and the underlying server-side SQL buffer or column is of `DATE` type, then the session time zone is assigned.

## 5.4.4 Data Conversion for BINARY_FLOAT and BINARY_DOUBLE in OCI

Shows the supported conversions between internal numeric data types and all relevant external types.

Table 5-8 shows the supported conversions between internal numeric data types 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). An (I/O) implies that the conversion is valid for input and output (binds and defines).

**Table 5-8 Data Conversion for External Data Types to Internal Numeric Data Types**

| External Types/Internal Types | BINARY_FLOAT | BINARY_DOUBLE |
|---|---|---|
| `VARCHAR` | I/O[1] | I/O |
| `VARCHAR2` | I/O | I/O |
| `NUMBER` | I/O | I/O |
| `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 |

[1] An (I/O) implies that the conversion is valid for input and output (binds and defines)

Table 5-9 shows the supported conversions between all relevant internal types and numeric 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). An (I/O) implies that the conversion is valid for input and output (binds and defines).

**Table 5-9 Data Conversions for Internal to External Numeric Data Types**

| Internal Types/External Types | Native Float | Native Double |
|---|---|---|
| `VARCHAR2` | I/O[1] | I/O |
| `NUMBER` | I/O | I/O |
| `LONG` | I[2] | I |
| `CHAR` | I/O | I/O |
| `BINARY_FLOAT` | I/O | I/O |

**Table 5-9 (Cont.) Data Conversions for Internal to External Numeric Data Types**

| Internal Types/External Types | Native Float | Native Double |
|---|---|---|
| `BINARY_DOUBLE` | I/O | I/O |

[1] An (I/O) implies that the conversion is valid for input and output (binds and defines)

[2] An (I) implies that the conversion is valid for input only (only for binds)

## 5.5 Typecodes

A unique typecode is associated with each Oracle Database type, whether scalar, collection, reference, or object type.

This typecode identifies the type, and is used by Oracle Database to manage information about object type attributes. This typecode system is designed to be generic and extensible. It is not tied to a direct one-to-one mapping to Oracle data types. 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` has 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 data types 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 data type 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 5-10 lists the possible values for an `OCITypeCode`. There is a value corresponding to each Oracle data type.

**Table 5-10 OCITypeCode Values and Data Types**

| Value | Data Type |
|---|---|
| `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 |

**Table 5-10 (Cont.) OCITypeCode Values and Data Types**

| Value | Data Type |
|---|---|
| `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` | Collection |
| `OCI_TYPECODE_BOOLEAN`[1] | Boolean |
| `OCI_TYPECODE_RECORD`[1] | Record |
| `OCI_TYPECODE_ITABLE`[1] | Index-by BINARY_INTEGER |
| `OCI_TYPECODE_INTEGER`[1] | PLS_INTEGER or BINARY_INTEGER |

[1] This type is a PL/SQL type only.

This section includes the following topic: Relationship Between SQLT and OCI_TYPECODE Values.

## 5.5.1 Relationship Between SQLT and OCI_TYPECODE Values

Oracle Database recognizes two different sets of data type 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 data type in a bind or define operation, enabling you to control data conversions between Oracle Database 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_SIGNED8`, `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 5-11 illustrates the mappings between `SQLT` and `OCI_TYPECODE` types.

**Table 5-11 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` |
| `BOOLEAN`[1] | `OCI_TYPECODE_BOOLEAN` | `SQLT_BOL` |
| `CHAR` | `OCI_TYPECODE_CHAR` (n) | `SQLT_AFC`(n)[2] |
| `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)[3] |
| `DECIMAL` | `OCI_TYPECODE_DECIMAL` (p) | `SQLT_NUM` (p, 0)[4] |
| `DOUBLE` | `OCI_TYPECODE_DOUBLE` | `SQLT_FLT` (8) |
| `BINARY_FLOAT` | `OCI_TYPECODE_BFLOAT` | `SQLT_BFLOAT` |
| `BINARY_DOUBLE` | `OCI_TYPECODE_BDOUBLE` | `SQLT_BDOUBLE` |
| `INDEX-BY BINARY_INTEGER`[1] | `OCI_TYPECODE_ITABLE` | `SQLT_NTY` |
| `INTEGER` | `OCI_TYPECODE_INTEGER` | `SQLT_INT` (i)[5] |
| `NUMBER` | `OCI_TYPECODE_NUMBER` (p, s) | `SQLT_NUM` (p, s)[6] |
| `OCTET` | `OCI_TYPECODE_OCTET` | `SQLT_INT` (1) |
| `PLS_INTEGER` or `BINARY_INTEGER`[1] | `OCI_TYPECODE_PLS_INTEGER` | `SQLT_INT` |
| `POINTER` | `OCI_TYPECODE_PTR` | <NONE> |
| `RAW` | `OCI_TYPECODE_RAW` | `SQLT_LVB` |
| `REAL` | `OCI_TYPECODE_REAL` | `SQLT_FLT` (4) |
| `REF` | `OCI_TYPECODE_REF` | `SQLT_REF` |

**Table 5-11 (Cont.) OCI_TYPECODE to SQLT Mappings**

| Oracle Type System Typename | Oracle Type System Type | Equivalent SQLT Type |
|---|---|---|
| `RECORD`[1] | `OCI_TYPECODE_RECORD` | `SQLT_NTY` |
| `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) |
| `SIGNED(32)` | `OCI_TYPECODE_SIGNED32` | `SQLT_INT` (4) |
| `SMALLINT` | `OCI_TYPECODE_SMALLINT` | `SQLT_INT` (i)[5] |
| `TABLE`[7] | `OCI_TYPECODE_TABLE` | <NONE> |
| `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`[7] | `OCI_TYPECODE_VARRAY` | <NONE> |
| `VARCHAR` | `OCI_TYPECODE_VARCHAR` (n) | `SQLT_CHR` (n)[2] |
| `VARCHAR2` | `OCI_TYPECODE_VARCHAR2` (n) | `SQLT_VCS` (n)[2] |

[1] This type is a PL/SQL type only.

[2] n is the size of the string in bytes.

[3] These are floating-point numbers, the precision is given in terms of binary digits. b is the precision of the number in binary digits.

[4] This is equivalent to a NUMBER with no decimal places.

[5] i is the size of the number in bytes, set as part of an OCI call.

[6] p is the precision of the number in decimal digits; s is the scale of the number in decimal digits.

[7] Can only be part of a named collection type.

## 5.6 Definitions in oratypes.h

Describes the contents of the `oratypes.h` header file.

Throughout this guide there are references to data types 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 that you are using.

**Note:**

The use of the data types in `oratypes.h` is the only supported means of supplying parameters to OCI.

# 6

# Using SQL Statements in OCI

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

This chapter contains these topics:

- Overview of SQL Statement Processing
- About Preparing Statements
- About Binding Placeholders in OCI
- About Executing Statements
- About Describing Select-List Items
- About Defining Output Variables in OCI
- About Fetching Results
- About Using Scrollable Cursors in OCI

## 6.1 Overview of SQL Statement Processing

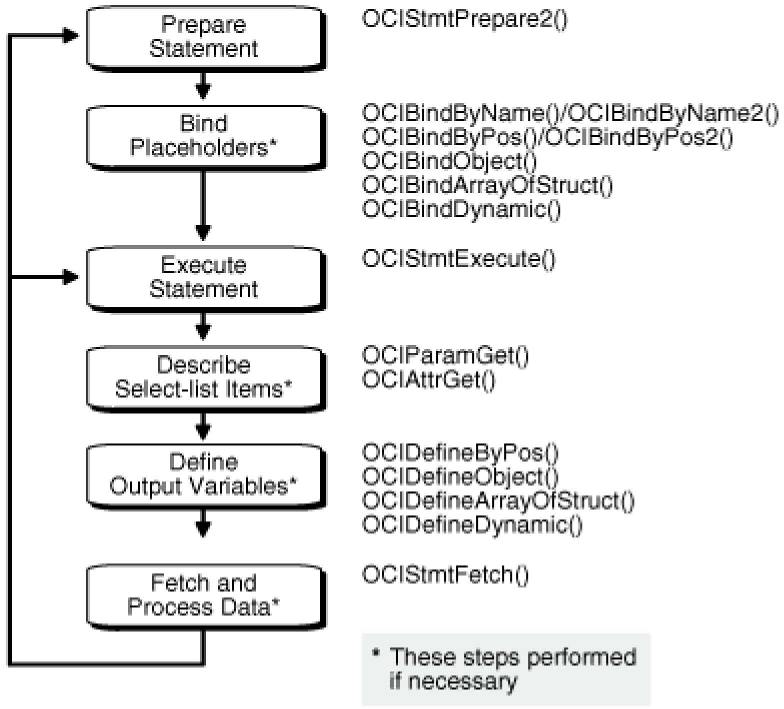
One of the most common tasks of an OCI program is to accept and process SQL statements.

Chapter 3 “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.

This section outlines the specific steps involved in accepting and processing SQL statements.

Once you have allocated the necessary handles and connected to an Oracle database, follow the steps illustrated in Figure 6-1.

**Figure 6-1 Steps in Processing SQL Statements**


Prepare Statement
OCIStmtPrepare2()
Bind Placeholders*
OCIBindByName()/OCIBindByName2()
OCIBindByPos()/OCIBindByPos2()
OCIBindObject()
OCIBindArrayOfStruct()
OCIBindDynamic()
Execute Statement
OCIStmtExecute()
Describe Select-list Items*
OCIParamGet()
OCIAttrGet()
Define Output Variables*
OCIDefineByPos()
OCIDefineObject()
OCIDefineArrayOfStruct()
OCIDefineDynamic()
Fetch and Process Data*
OCIStmtFetch()
* These steps performed if necessary

1. Prepare the statement. Define an application request using `OCIStmtPrepare2()`. `OCIStmtPrepare2()` is an enhanced version of `OCIStmtPrepare()` that was introduced to support statement caching. Beginning with Oracle Database 12c Release 2 (12.2), `OCIStmtPrepare()` is deprecated.
2. Bind placeholders, if necessary. For DML statements and queries with input variables, perform one or more of the following bind calls to bind the address of each input variable (or PL/SQL output variable) or array to each placeholder in the statement.
   - `OCIBindByPos2()` or `OCIBindByPos()`
   - `OCIBindByName2()` or `OCIBindByName()`
   - `OCIBindObject()`
   - `OCIBindDynamic()`
   - `OCIBindArrayOfStruct()`
3. Execute the statement by calling `OCIStmtExecute()`. For DDL statements, no further steps are necessary.
4. Describe the select-list items, if necessary, using`OCIParamGet()` and `OCIAttrGet()`. This is optional step is not required if the number of select-list items and the attributes of each item (such as its length and data type) are known at compile time.
5. Define output variables, if necessary. For queries, perform one or more define calls to `OCIDefineByPos2()` or `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 did this when you bound the data.

6. Fetch the results of the query, if necessary, by calling `OCIStmtFetch2()`.

After these steps have been completed, the application can free allocated handles and then detach from the server, or it may process additional statements.

> **Note:**
>
> OCI programs no longer require an explicit parse step. If a statement must be parsed, that step occurs upon execution, meaning that release 8.0 or later applications must issue an execute command for both DML and DDL statements.

The following sections describe each step in detail.

> **Note:**
>
> Some variation in the order of steps is possible. For example, it is possible to do the define step before the execute step if the data types and lengths of returned values are known at compile time.

Additional steps beyond those listed earlier may be required if your application must 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
- OCI Programming Basics
- OCIStmtPrepare2() or OCIStmtPrepare()
- OCIBindByPos2() or OCIBindByPos()
- OCIBindByName2() or OCIBindByName()
- OCIBindObject()
- OCIBindDynamic()
- OCIBindArrayOfStruct()
- OCIStmtExecute()
- OCIParamGet()
- OCIAttrGet()
- OCIDefineByPos2() or OCIDefineByPos()
- OCIDefineObject()
- OCIDefineDynamic()
- OCIDefineArrayOfStruct()
- OCIStmtFetch2()

## 6.2 About 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 `OCIStmtPrepare2()` call and passes to this call 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 6-1.

**Table 6-1 OCI_ATTR_STMT_TYPE Values and Statement Types**

| Attribute Value | Statement Type |
|---|---|
| `OCI_STMT_SELECT` | `SELECT` statement |
| `OCI_STMT_UPDATE` | `UPDATE` statement |

**Table 6-1 (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) |
| `OCI_STMT_CALL` | `CALL`... (PL/SQL) |
| `OCI_STMT_MERGE` | `MERGE`... (PL/SQL) |

This section includes the following topic: About Using Prepared Statements on Multiple Servers

> **See Also:**
>
> - OCIStmtPrepare2()
> - OCIAttrGet()
> - About Using PL/SQL in an OCI Program

## 6.2.1 About 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 must be executed against multiple servers to retrieve information for display. 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 before reassociating the prepared statement with the next server.

> **Note:**
>
> If a prepared statement must be reexecuted frequently on the same server, it is more efficient to prepare a new statement for another service context.

# 6.3 About 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 Database 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 must 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 in 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, the database gets the data that your program placed in the input variables or bind variables and passes it to the server with the SQL statement.

Binding is used for both input and output variables in nonquery operations. In Example 6-1, the variables `empno_out`, `ename_out`, `job_out`, `sal_out`, and `deptno_out` should be bound. These are outbinds (as opposed to regular inbinds).

**Example 6-1    Binding Both Input and Output Variables in Nonquery Operations**

```
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)
```

This section includes the following topic: Rules for Placeholders

**See Also:**

Binding and Defining in OCI for detailed information about implementing bind operations

## 6.3.1 Rules for Placeholders

Lists and describes the rules for forming placeholders.

The rules for forming placeholders are as follows:

- The first character is a colon (":").
- The colon is followed by a combination of underscore ("_"), A to Z, a to z, or 0 to 9. However, the first character following the colon cannot be an underscore.
- The letters must be only from the English alphabet, and only the first 30 characters after the colon are significant. The name is case-insensitive.
- The placeholder can consist of only digits after the colon. If it is only digits, the placeholder must be less than 65536. If the name starts with a digit, then only digits are allowed.
- The hyphen ("-") is not allowed.

# 6.4 About Executing Statements

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

When an OCI application executes a query, it receives from the Oracle database data 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 `OCIStmtFetch2()`.

For nonqueries, the number of times the statement is executed during array operations equals `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 are inserted in a single execute call when `rowoff` is zero. If `rowoff` is set to 2, only 8 items are inserted.

This section includes the following topics:

- Execution Snapshots
- Execution Modes of OCIStmtExecute()

**See Also:**

- OCIStmtExecute()
- OCIStmtFetch2()
- About Defining Output Variables in OCI

## 6.4.1 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 both calls are using the same snapshot.

The data type of both the `snap_out` and `snap_in` parameter is `OCISnapshot`. OCISnapshot is an OCI snapshot descriptor that is allocated with the `OCIDescriptorAlloc()` function.

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.

**See Also:**

- OCIStmtExecute()
- OCIDescriptorAlloc()
- OCI Descriptors

## 6.4.2 Execution Modes of OCIStmtExecute()

You can specify a number of modes for the `OCIStmtExecute()` call.

This section describes the OCIStmtExecute() call. See `OCIStmtExecute()` for other values of the parameter `mode`.

This section includes the following topics:

- Using Batch Error Mode
- Example of Batch Error Mode

**See Also:**

OCIStmtExecute()

### 6.4.2.1 Using 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 terminates, and OCI returns an error. Any rows remaining in the array are ignored. The application must then reexecute the remainder of the array, and go through the whole process again if it encounters more errors, which causes 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 if there are 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 reexecute any DML operations that 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 release 8.1 or later OCI libraries running against a release 8.1 or later server. Applications must also be recoded to account for the new program logic described in this section.

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, as shown in the following code example.

   Calling OCIAttrGet() to Retrieve the Number of Errors Encountered During an Array DML Operation

   ```
   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. 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, after the `num_errs` amount has been retrieved, the application can issue the following calls shown in the following code example.

   Retrieving Information About Each Error Following an Array DML Operation

   ```
   OCIError errhndl, errhp2;
   for (i=0; i<num_errs; i++)
   {
      OCIParamGet(errhp, OCI_HTYPE_ERROR, errhp2, (void **)&errhndl, i);
      OCIAttrGet(errhndl, OCI_HTYPE_ERROR, &row_offset, 0,
          OCI_ATTR_DML_ROW_OFFSET, errhp2);
      OCIErrorGet(..., errhndl, ...);
   ```

   Following this operation, the application can correct the bind information for the appropriate entry in the array using the diagnostic information retrieved from the batched error. Once the appropriate bind buffers are corrected or updated, the application can reexecute the associated DML statements.

   Because it cannot be determined at compile time which rows in the first execution may cause errors, the binds for the subsequent DML should be done dynamically by passing in the appropriate buffers at run time. The bind buffers used in the array binds done on the first DML operation can be reused.

**See Also:**

- OCIStmtExecute()
- OCIAttrGet()
- OCIParamGet()
- OCIErrorGet()

## 6.4.2.2 Example of Batch Error Mode

Shows how the batch error execution mode might be used.

Example 6-2 shows an example of how the batch error execution mode might be used. In this example, assume that you have an application that inserts five rows (with two columns, of types `NUMBER` and `CHAR`) into a table. Furthermore, assume that only two rows (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` statements, respectively.

In Example 6-2, `OCIBindDynamic()` is used with a callback because the user does not know at compile time what rows may 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 must be updated or corrected. The same bind buffers can be shared between the `INSERT` and the `UPDATE` statement executions.

**Example 6-2    Using Batch Error Execution Mode**

```
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,(void *)&number[0],
     sizeof(number[0]),SQLT_INT,(void *)0, (ub2 *)0,(ub2 *)0,
            0,(ub4 *)0,OCI_DEFAULT);
OCIBindByPos (stmtp1,&bindp1[1],errhp,2,(void *)&grade[0],
     sizeof(grade[0]),SQLT_CHR,(void *)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( (void *)envhp, (void **)&errhndl[i],
      (ub4) OCI_HTYPE_ERROR, 0, (void *) 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,(void *)0,sizeof(grade[0]),SQLT_INT,
     (void *)0, (ub2 *)0,(ub2 *)0,0,(ub4 *)0,OCI_DATA_AT_EXEC);
OCIBindByPos (stmtp2,&bindp2[1],errhp,2,(void *)0,sizeof(number[0]),SQLT_DAT,
     (void *)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 the callback 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);
...
```

**See Also:**

OCIBindDynamic()

## 6.5 About 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 data types and column lengths of the select-list items. This information is necessary to define output variables that may 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* 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 statement execution.

An *explicit describe* 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 before execution by specifying `OCI_DESCRIBE_ONLY` as the mode of `OCIStmtExecute()`, which does not execute the statement, but returns the select-list description. For performance reasons, Oracle recommends that applications use the implicit describe, which 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 data types is retrieved by reading handle attributes.

This section includes the following topics:

- Implicit Describe
- Explicit Describe of Queries

**See Also:**

- Describing Schema Metadata for information about using OCIDescribeAny() to obtain metadata pertaining to schema objects
- OCIStmtExecute()
- OCIDescribeAny()

## 6.5.1 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 data type and maximum size of the parameter.

The sample code in Example 6-3Example 6-3 shows a loop that retrieves the column names and data types corresponding to a query following query execution. The query was associated with the statement handle by a prior call to `OCIStmtPrepare2()`.

The `checkerr()` function in Example 6-3 is used for error handling. The complete listing can be found in the first sample application in 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.

**Example 6-3 Implicit Describe - Select List Is Available as an Attribute of the Statement Handle**

```
...
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, OCIStmtPrepare2(svchp, &stmthp, errhp, (OraText *)sqlstmt,
                    (ub4)strlen((char *)sqlstmt), NULL, 0,
                    (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((void *)stmthp, OCI_HTYPE_STMT, errhp,
               (void **)&mypard, (ub4) counter);

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

while (parm_status == OCI_SUCCESS) {
   /* Retrieve the data type attribute */
   checkerr(errhp, OCIAttrGet((void*) mypard, (ub4) OCI_DTYPE_PARAM,
           (void*) &dtype,(ub4 *) 0, (ub4) OCI_ATTR_DATA_TYPE,
           (OCIError *) errhp  ));

   /* Retrieve the column name attribute */
   col_name_len = 0;
   checkerr(errhp, OCIAttrGet((void*) mypard, (ub4) OCI_DTYPE_PARAM,
           (void**) &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((void*) mypard, (ub4) OCI_DTYPE_PARAM,
           (void*) &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((void*) mypard, (ub4) OCI_DTYPE_PARAM,
               (void*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_SIZE,
               (OCIError *) errhp  ));
   else
       /* Retrieve the column width in bytes */
       checkerr(errhp, OCIAttrGet((void*) mypard, (ub4) OCI_DTYPE_PARAM,
               (void*) &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((void *)stmthp, OCI_HTYPE_STMT, errhp,
          (void **)&mypard, (ub4) counter);
} /* while */
...
```

**See Also:**

- OCIParamGet()
- OCIAttrGet()
- OCIStmtPrepare2()
- OCIArrayDescriptorAlloc()
- Parameter Attributes for a list of the specific attributes of the parameter descriptor that may be read by OCIArrayDescriptorAlloc()

## 6.5.2 Explicit Describe of Queries

You can describe a query explicitly before execution by specifying `OCI_DESCRIBE_ONLY` as the mode of `OCIStmtExecute()`

This does not execute the statement, but returns the select-list description.

**Note:**

To maximize performance, Oracle recommends that applications execute the statement in default mode and use the implicit describe that accompanies the execution.

The code in Example 6-4 demonstrates the use of explicit describe in a select list to return information about columns.

**Example 6-4 Explicit Describe - Returning the Select-List Description for Each Column**

```
...
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, OCIStmtPrepare2(svchp, &stmthp, errhp, (OraText *)sqlstmt,
                    (ub4)strlen((char *)sqlstmt), NULL, 0,
                    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

/* initialize svchp, stmthp, 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((void *)stmthp, OCI_HTYPE_STMT, (void *)&numcols,
                        (ub4 *)0, OCI_ATTR_PARAM_COUNT, errhp));

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

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

**See Also:**

OCIStmtExecute()

## 6.6 About 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 from which to retrieve data. 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 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:**

Binding and Defining in OCI

## 6.7 About Fetching Results

If an OCI application has processed a query, it is typically necessary to fetch the results with `OCIStmtFetch2()` after the statement has completed execution.

The `OCIStmtFetch2()` function supports *scrollable cursors*.

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

This section includes the following topics:

- About Fetching LOB Data
- About Setting Prefetch Count

**See Also:**

- About Using Scrollable Cursors in OCI
- OCIStmtFetch2()
- OCIStmtExecute()
- These statements mentioned previously fetch data associated with the sample code in Steps Used in OCI Defining. See that example for more information.
- Overview of Defining in OCI for information about defining output variables

## 6.7.1 About 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` functions.

**See Also:**

- LOB and BFILE Operations for more information about working with LOB locators in OCI
- About Defining LOB Output Variables for usage and examples of selecting LOB data without the use of locators

## 6.7.2 About Setting Prefetch Count

To minimize server round-trips and optimize performance, 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 can fit into that much memory.

When both of these attributes are set, OCI prefetches rows up to the `OCI_ATTR_PREFETCH_ROWS` limit unless the `OCI_ATTR_PREFETCH_MEMORY` limit is reached, in which case OCI returns as many rows as can fit in a buffer of size `OCI_ATTR_PREFETCH_MEMORY`.

By default, prefetching is turned on, and OCI fetches one extra row, except when prefetching cannot be supported for a query (see the note that follows). To turn prefetching off, set both the `OCI_ATTR_PREFETCH_ROWS` and `OCI_ATTR_PREFETCH_MEMORY` attributes to zero.

If both `OCI_ATTR_PREFETCH_ROWS` and `OCI_ATTR_PREFETCH_MEMORY` attributes are explicitly set, OCI uses the tighter of the two constraints to determine the number of rows to prefetch.

To prefetch exclusively based on the memory constraint, set the `OCI_ATTR_PREFETCH_MEMORY` attribute and be sure to disable the `OCI_ATTR_PREFETCH_ROWS` attribute by setting it to zero (to override the default setting of 1 row).

To prefetch exclusively based on the number of rows constraint, set the `OCI_ATTR_PREFETCH_ROWS` attribute and disable the `OCI_ATTR_PREFETCH_MEMORY` attribute by setting it to zero (if it was ever explicitly set to a non-zero value).

Prefetching is possible for `REF CURSOR`s and nested cursor columns. By default, prefetching is not turned on for `REF CURSOR`s. To turn on prefetching for `REF CURSOR`s, set the `OCI_ATTR_PREFETCH_ROWS` or `OCI_ATTR_PREFETCH_MEMORY` attribute before fetching rows from the `REF CURSOR`. When a `REF CURSOR` is passed multiple times between an OCI application and PL/SQL and fetches on the `REF CURSOR` are done in OCI and in PL/SQL, the rows prefetched by OCI (if enabled) cause the application to behave as if out-of-order rows are being fetched in PL/SQL. In such situations, OCI prefetch should not be enabled on `REF CURSOR`s.

**Note:**

Prefetching is not in effect if `LONG`, LOB or Opaque Type columns (such as `XMLType`) are part of the query.

**See Also:**

- Statement Handle Attributes
- OCIAttrSet()
- OCIStmtExecute()
- OCIStmtFetch2()

## 6.8 About Using 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 nonsequentially. In the latter case, the cursor is known as a *scrollable cursor.*

A scrollable cursor supports forward and backward access into the result set from a given position, by 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 nth row in the result set, or the nth 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 Database does not support DML operations on scrollable cursors. A cursor cannot be made scrollable if the `LONG` data type 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 their functionality, because they use more server resources and can have greater response times than nonscrollable cursors.

The `OCIStmtExecute()` call has an execution mode for scrollable cursors, `OCI_STMT_SCROLLABLE_READONLY`. The default for statement handles is nonscrollable, 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 only by using `OCIAttrGet()`. This attribute cannot be set by the application; it indicates the current position in the result set.

For nonscrollable cursors, `OCI_ATTR_ROW_COUNT` is the total number of rows fetched into the user buffers with the `OCIStmtFetch2()` calls since this statement handle was executed. Because nonscrollable cursors are forward sequential only, `OCI_ATTR_ROW_COUNT` also represents the highest row number detected by the application.

Beginning with Oracle Database Release 12.1, using the attribute `OCI_ATTR_UB8_ROW_COUNT` is preferred to using the attribute `OCI_ATTR_ROW_COUNT` if row count values can exceed the value of `UB4MAXVAL` for an OCI application.

For scrollable cursors, `OCI_ATTR_ROW_COUNT` represents the maximum (absolute) row number fetched into the user buffers. Because the application can arbitrarily position the fetches, this does not have to be the total number of rows fetched into the user's buffers since the (scrollable) statement was executed.

The attribute `OCI_ATTR_ROWS_FETCHED` on the statement handle represents the number of rows that were successfully fetched into the user's buffers in the last fetch call or execute. It works for both scrollable and nonscrollable 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 nonscrollable 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 canceled.

This section includes the following topics:

- About Increasing Scrollable Cursor Performance
- Example of Access on a Scrollable Cursor

> **See Also:**
>
> - OCIStmtExecute()
> - OCIAttrGet()
> - OCIStmtFetch2()
> - About Setting Prefetch Count

## 6.8.1 About 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.

**See Also:**

- OCIStmtExecute()
- OCIStmtFetch2()

## 6.8.2 Example of Access on a Scrollable Cursor

Shows the use of a scrollable cursor.

Assume that a result set is returned by the following SQL query, and that the table `EMP` has 14 rows:

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

One use of scrollable cursors is shown in Example 6-5.

**Example 6-5    Access on a Scrollable Cursor**

```
...
/* 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));
}
...
```

# 7

# Binding and Defining in OCI

This chapter describes binding and defining in OCI.

This chapter contains these topics:

- Overview of Binding in OCI
- Advanced Bind Operations in OCI
- Overview of Defining in OCI
- Advanced Define Operations in OCI
- About Binding and Defining Arrays of Structures in OCI
- About Binding and Defining Multiple Buffers
- DML with a RETURNING Clause in OCI
- Character Conversion in OCI Binding and Defining
- PL/SQL REF CURSORs and Nested Tables in OCI
- Natively Describe and Bind All PL/SQL Types Including Package Types
- Runtime Data Allocation and Piecewise Operations in OCI

## 7.1 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, and other issues involved in binding, defining, and character conversions.

For example, given the `INSERT` statement:

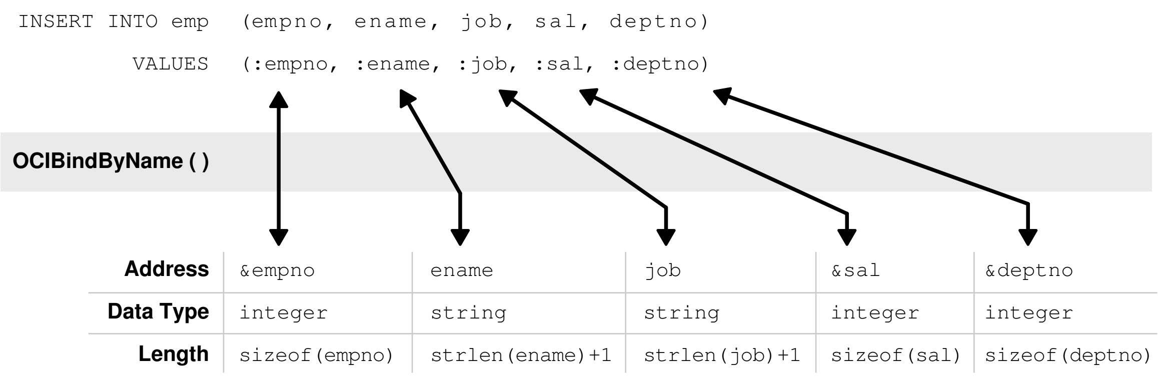
```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

Then given 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 data type and length of the program variables, as illustrated in Figure 7-1.

**Figure 7-1 Using OCIBindByName() to Associate Placeholders with Program Variables**



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

| | | | | | |
|---|---|---|---|---|---|
| **Address** | &empno | ename | job | &sal | &deptno |
| **Data Type** | integer | string | string | integer | integer |
| **Length** | sizeof(empno) | strlen(ename)+1 | strlen(job)+1 | sizeof(sal) | sizeof(deptno) |

If you change only the value of a bind variable, it is not necessary to rebind it 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 reexecute a statement that references the variable without rebinding.

> **Note:**
>
> At the interface level, all bind variables are considered at least `IN` and must be properly initialized. If the variable is a pure `OUT` bind variable, you can set the variable to 0. You can also provide a `NULL` indicator and set that indicator to -1 (`NULL`).

In the Oracle database, data types have been implemented for named data types, `REF`s and LOBs, and they can be bound as placeholders in a SQL statement.

> **Note:**
>
> For opaque data types (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)`).
>
> This section includes the following topics:
>
> - Named Binds and Positional Binds
> - OCI Array Interface
> - About Binding Placeholders in PL/SQL
> - Steps Used in OCI Binding
> - PL/SQL Block in an OCI Program

**See Also:**

Steps Used in OCI Binding for the code that implements this example

## 7.1.1 Named Binds and Positional Binds

In a named bind, each placeholder in the statement has a name associated with it, while in a positional bind, the placeholders are referred to by their position in the statement rather than by their names.

The SQL statement in Figure 7-1 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()` or `OCIBindByName2()` 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 by their names. For binding purposes, an association is made between an input value and the position of the placeholder, using the `OCIBindByPos()` or `OCIBindByPos2()` call.

To use 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()` or `OCIBindByPos2()` and passing the position number of the placeholder in the `position` parameter. For example, the `:empno` placeholder would be bound by calling `OCIBindByPos()` or `OCIBindByPos2()` with a position of 1, `:ename` with a position of 2, and so on.

In 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()` or `OCIBindByName2()` to bind the `:some_value` placeholder by name. In this case, all bind placeholders for `:some_value` get assigned the same value as provided by the `OCIBindByName()` or `OCIBindByName2()` call.

Now consider the case where a 6th placeholder is added that is a duplicate. For example, add `:ename` as the 6th placeholder in the first previous example:

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

If you are using the `OCIBindByName()` or `OCIBindByName2()` call, just one bind call suffices to bind both occurrences of the `:ename` placeholder. All occurrences of `:ename` in the statement will get bound to the same value. Moreover, if new bind placeholders get added as a result of which bind positions for existing bind placeholders change, you do not need to change your existing bind calls in order to update bind positions. This is a distinct advantage in using the `OCIBindByName()` or `OCIBindByName2()` call if your program evolves to add more bind variables in your statement text.

If you are using the `OCIBindByPos()` or `OCIBindByPos2()` call, however, you have increased flexibility in terms of binding duplicate bind-parameters separately, if you need it. You have the option of binding any of the duplicate occurrences of a bind parameter separately. Any unbound duplicate occurrences of a parameter inherit the value from the first occurrence of the bind parameter with the same name. The first occurrence must be explicitly bound.

In the context of SQL statements, the position *n* indicates the bind parameter at the *n*th position. However, in the context of PL/SQL statements, `OCIBindByPos()` or `OCIBindByPos2()` has a different interpretation for the position parameter: the position *n* in the bind call indicates a binding for the *n*th unique parameter name in the statement when scanned left to right.

Using the previous example again and the same SQL statement text, if you want to bind the 6th position separately, the `:ename` placeholder would be bound by calling `OCIBindByPos()` or `OCIBindByPos2()` with a position of 6. Otherwise, if left unbound, `:ename` would inherit the value from the first occurrence of the bind parameter with the same name, in this case, from `:ename` in position 2.

**See Also:**

- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()

## 7.1.2 OCI Array Interface

You can pass data to the Oracle database 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 inserts 10 rows into the database. Calling `OCIStmtExecute()` 10 times with single values results in 10 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.

Beginning with Oracle Database 12*c* Release 2 (12.2), support is added for Hybrid Columnar Compression (HCC) with conventional DMLs, so HCC can be used during array inserts with OCI. HCC conventional array inserts are only supported for HCC tables on ASSM tablespaces. .

> **Note:**
>
> When you use 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 billion -1 (3,999,999,999). However, if you use `ub8` instead of `ub4`, this increases the maximum number of rows allowed in an array DML statement to be more than 4 billion rows.

> **See Also:**
>
> - OCIStmtExecute()
> - About Table Compression in *Oracle Database Administrator's Guide* for information about how to configure HCC

## 7.1.3 About 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 `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()` 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, when 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 by using bind calls.

For example, consider the following PL/SQL block:

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

In this block, you would use `OCIBindByName()` or `OCIBindByName2()` 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:**

- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- Information for Named Data Type and REF Binds for more information about binding PL/SQL placeholders

## 7.1.4 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 `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`.

Once the bind is complete, the OCI library detects where to find the input data or where to put the 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 in Example 7-1 shows handle allocation and binding for each placeholder in a SQL statement.

**Note:**

The `checkerr()` function evaluates the return code from an OCI application. The code for the function is in the Example for `OCIErrorGet()`.

**Example 7-1 Handle Allocation and Binding for Each Placeholder in a SQL Statement**

```
...
/* The SQL statement, associated with stmthp (the statement handle)
by calling OCIStmtPrepare2() */
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, (void *) 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, (void *)
    &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,
    (void *) &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,
    (void *) 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,
    (void *) 0, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0,OCI_DEFAULT));
```

**See Also:**

- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIErrorGet()

## 7.1.5 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 that 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.

Example 7-2 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 data types or execution modes.

**Example 7-2    Defining a PL/SQL Statement to Be Used in OCI**

```
/* 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;
  OCISession *usrhp = (OCISession *)NULL;
...
/* attach to Oracle database, and perform necessary initializations
and authorizations */
...
      /* prepare the statement request, passing the PL/SQL text
        block as the statement to be prepared */
checkerr(errhp, OCIStmtPrepare2(svchp, &stmthp, errhp, (text *) give_raise, (ub4)
      strlen(give_raise), NULL, 0, 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, (void *) 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, (void *) 0,
             (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

      /* remember that PL/SQL OUT variables are bound, not defined */

checkerr( errhp, OCIBindByName(stmthp, &bnd3p, errhp, (text *) ":new_salary",
             -1, (ub1 *) &new_sal,
             (sword) sizeof(new_sal), SQLT_INT, (void *) 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);

OCIStmtRelease(stmthp, errhp, NULL, 0, OCI_DEFAULT);
}
```

> **See Also:**
>
> Why a PL/SQL OUT variable must be bound and not defined is explained in About Defining PL/SQL Output Variables and in Information for Named Data Type and REF Defines, and PL/SQL OUT Binds.

## 7.2 Advanced Bind Operations in OCI

More advanced bind operations include multistep binds, and binds of named data types and `REF`s.

About Binding Placeholders in OCI discussed how a basic bind operation is performed to create an association between a placeholder in a SQL statement and a program variable by using `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`. This section covers more advanced bind operations, including multistep binds, and binds of named data types and `REF`s.

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

The following sections describe these special cases, and the information about binding is summarized in Table 7-1.

**Table 7-1 Information Summary for Bind Types**

| Type of Bind | Bind Data Type | Notes |
|---|---|---|
| Scalar | Any scalar data type | Bind a single scalar using `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`. |
| Array of scalars | Any scalar data type | Bind an array of scalars using `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`. |
| Named data type | `SQLT_NTY` | Includes records and collections<br>Two bind calls are required:<br>• `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`<br>• `OCIBindObject()` |
| Boolean | `SQLT_BOL` | Bind a Boolean using `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`. |
| `REF` | `SQLT_REF` | Two bind calls are required:<br>• `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`<br>• `OCIBindObject()` |
| LOB<br>`BFILE` | `SQLT_BLOB`<br>`SQLT_CLOB` | Allocate the LOB locator using `OCIDescriptorAlloc()`, and then bind its address, `OCILobLocator **`, with `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`, by using one of the LOB data types. |
| Array of structures or static arrays | Varies | Two bind calls are required:<br>• `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`<br>• `OCIBindArrayOfStruct()` |

**Table 7-1 (Cont.) Information Summary for Bind Types**

| Type of Bind | Bind Data Type | Notes |
|---|---|---|
| Piecewise insert | Varies | `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()` 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` data type. |

This section includes the following topics:

- About Binding LOBs
- About Binding in OCI_DATA_AT_EXEC Mode
- About Binding REF CURSOR Variables

**See Also:**

- Named Data Type Binds for information about binding named data types (objects)
- About Binding REFs
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIBindObject()
- OCIDescriptorAlloc()
- OCIBindArrayOfStruct()
- OCIBindDynamic()

## 7.2.1 About 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.

This section includes the following topics:

- Binding LOB Locators
- About Binding LOB Data

### 7.2.1.1 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, suppose that an application has prepared this SQL statement where `one_lob` is a bind variable corresponding to a LOB column:

```
INSERT INTO some_table VALUES (:one_lob)
```

Then your application makes the following declaration:

```
OCILobLocator * one_lob;
```

Then the calls in Example 7-3 would be used to bind the placeholder and execute the statement:

You can also insert an array using the same SQL `INSERT` statement. In this case, the application would include the code shown in Example 7-4.

You must allocate descriptors with the `OCIDescriptorAlloc()` function before they can be used. In 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.

**Example 7-3    Binding the Placeholder and Executing the Statement to Insert a Single Locator**

```
/* initialize single locator */
one_lob = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
...
/* pass the address of the locator */
OCIBindByName(...,(void *) &one_lob,... SQLT_CLOB, ...);
OCIStmtExecute(...,1,...)                /* 1 is the iters parameter */
```

**Example 7-4    Binding the Placeholder and Executing the Statement to Insert an Array of Locators**

```
OCILobLocator * lob_array[10];
...
for (i=0; i<10, i++)
    lob_array[i] = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
                                     /* initialize array of locators */
...
OCIBindByName(...,(void *) lob_array,...);
OCIStmtExecute(...,10,...);              /* 10 is the iters parameter */
```

This section includes the following topic: Restrictions on Binding LOB Locators

**See Also:**

OCIDescriptorAlloc()

### 7.2.1.1.1 Restrictions on Binding LOB Locators

What are the restrictions on binding LOB locators.

Observe the following restrictions when you bind 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 file name, by using `OCILobFileSetName()` before issuing the `INSERT` or `UPDATE` statement.

> **See Also:**
>
> - LOB and BFILE Operations for more information about the OCI LOB functions
> - OCILobFileSetName()

### 7.2.1.2 About Binding LOB Data

Oracle Database allows nonzero binds for `INSERT`s and `UPDATE`s of any size LOB.

So you can bind data into a LOB column using `OCIBindByPos()` or `OCIBindByPos2()`, `OCIBindByName()` or `OCIBindByName2()`, and PL/SQL binds.

The bind of more than 4 kilobytes of data to a LOB column uses space from the temporary tablespace. Ensure 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 is extended automatically after the existing space is fully consumed. Use the following command to create an extendable temporary tablespace:

```
CREATE TABLESPACE ... AUTOEXTEND ON ... TEMPORARY ...;
```

This section includes the following topics:

- Restrictions on Binding LOB Data
- Examples of Binding LOB Data

> **See Also:**
>
> - OCIBindByPos() or OCIBindByPos2()
> - OCIBindByName() or OCIBindByName2()

#### 7.2.1.2.1 Restrictions on Binding LOB Data

What are the restrictions on binding LOB data.

Observe the following restrictions when you bind 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 Database does not allow binding of any length data to LOB columns.

- A special consideration applies on the maximum size of bind variables that are neither LONG or LOB, but that appear after any LOB or LONG bind variable in the SQL statement. You receive an `ORA-24816` error from Oracle Database if the maximum size for such bind variables exceeds 4000 bytes. To avoid this error, you must set `OCI_ATTR_MAXDATA_SIZE` to 4000 bytes for any such binds whose maximum size may exceed 4000 bytes on the server side after character set conversion. Alternatively, reorder the binds so that such binds are placed before any LONG or LOBs in the bind list.
- Oracle Database does not do implicit conversions, such as `HEX` to `RAW` or `RAW` to `HEX`, for data of size more than 4000 bytes. The PL/SQL code in the following code example illustrates this:

  Demonstrating Some Implicit Conversions That Cannot Be Done

  ```
  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 does not work because Oracle dpes not do implicit
    -- hex to raw conversion.
    insert into t (c2) values (text);

    -- The following does not work because Oracle does not do implicit
    -- raw to hex conversion.
    insert into t (c1) values (binbuf);

    -- The following does not work because you cannot 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 Database limits 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'));
  ```

**See Also:**

About Using the OCI_ATTR_MAXDATA_SIZE Attribute

### 7.2.1.2.2 Examples of Binding LOB Data

Shows some exampled of binding LOB data.

The following SQL statements are used in Example 7-5 through Example 7-12:

```
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);
```

**Example 7-5 Allowed: Inserting into C1, C2, and L Columns Up to 8000, 8000, and 2000 Byte-Sized Bind Variable Data Values, Respectively**

```
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, (void *)buffer, 8000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (void *)buffer, 8000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (void *)buffer, 2000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL,
                              (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

**Example 7-6 Allowed: Inserting into C1 and L Columns up to 2000 and 8000 Byte-Sized Bind Variable Data Values, Respectively**

```
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, (void *)buffer, 2000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (void *)buffer, 8000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL,
                              (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

### Example 7-7 Allowed: Updating C1, C2, and L Columns up to 8000, 8000, and 2000 Byte-Sized Bind Variable Data Values, Respectively

```
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, (void *)buffer, 8000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (void *)buffer, 8000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (void *)buffer, 2000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL,
                           (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

### Example 7-8 Allowed: Updating C1, C2, and L Columns up to 2000, 2000, and 8000 Byte-Sized Bind Variable Data Values, Respectively

```
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, (void *)buffer, 2000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (void *)buffer, 2000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (void *)buffer, 8000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL,
                            (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

### Example 7-9 Allowed: Piecewise, Callback, and Array Insert or Update Operations

```
void insert()
{
   /* Piecewise, callback and array insert/update operations similar to
    * the allowed regular insert/update operations are also allowed */
}
```

### Example 7-10 Not Allowed: Inserting More Than 4000 Bytes into Both LOB and LONG Columns Using the Same INSERT Statement

```
void insert()
{
   /* The following is NOT allowed because you cannot 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, (void *)buffer, 8000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (void *)buffer, 8000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL,
                               (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

**Example 7-11    Allowed: Inserting into the CT3 LOB Column up to 2000 Byte-Sized Bind Variable Data Values**

```
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, (void *)buffer, 2000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL,
                               (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

**Example 7-12    Not Allowed: Binding Any Length Data to a LOB Column in an Insert As Select Operation**

```
void insert()
{
   /* The following is NOT allowed because you cannot 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, (void *)buffer, 8000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL,
                               (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

## 7.2.2 About Binding in OCI_DATA_AT_EXEC Mode

If the `mode` parameter in a call to `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()` is set to `OCI_DATA_AT_EXEC`, an additional call to `OCIBindDynamic()` is necessary if the application uses the callback method for providing data at run time.

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 is 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 for more information about piecewise operations
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIBindDynamic()

### 7.2.3 About Binding REF CURSOR Variables

`REF CURSOR`s are bound to a statement handle with a bind data type of `SQLT_RSET`.

**See Also:**

PL/SQL REF CURSORs and Nested Tables in OCI

## 7.3 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 then you would normally 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 that you define can be arrays instead of scalar values.

Depending on the application, the define step can occur before or after an execute operation. If you know the data types of select-list items at compile time, the define can occur before the statement is executed. If your application is processing dynamic SQL statements entered by you at run time 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.

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 occur when data is returned to the application.

**Note:**

Output buffers must be 2-byte aligned.

The `dty` parameter of the `OCIDefineByPos()` or `OCIDefineByPos2()` call specifies the data type of the output variable. OCI can perform 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` data type on output.

This section includes the following topic: Steps Used in OCI Defining

**See Also:**

- Data Types for more information about data types and conversions
- About Describing Select-List Items
- OCIDefineByPos() or OCIDefineByPos2()

## 7.3.1 Steps Used in OCI Defining

A basic define is done with a position call, `OCIDefineByPos()` or `OCIDefineByPos2()`.

This step creates an association between a select-list item and an output variable. Additional define calls may be necessary for certain data types 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 reprepare or reexecute the SQL statement.

Example 7-13 shows a scalar output variable being defined following an execute and describe operation.

**See Also:**

- About Describing Select-List Items for an explanation of the describe step
- OCIDefineByPos() or OCIDefineByPos2()

**Example 7-13 Defining a Scalar Output Variable Following an Execute and Describe Operation**

```
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 does not exist.\n");
    return 0;
  }

   /* The next two statements describe the select-list item, dname, and
   return its length */
  checkerr(errhp, OCIParamGet((void *)stmthp, (ub4) OCI_HTYPE_STMT, errhp, (void
**)&parmdp, (ub4) 1));
  checkerr(errhp, OCIAttrGet((void*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (void*) &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, (void *) dept, (sb4) deptlen+1,
             SQLT_STR, (void *) 0, (ub2 *) 0,
             (ub2 *) 0, OCI_DEFAULT))
  {
    checkerr(errhp, status);
    return OCI_ERROR;
  }
```

# 7.4 Advanced Define Operations in OCI

This section covers advanced define operations, including multistep defines and defines of named data types and REFs.

In some cases, the define step requires additional calls than just a call to `OCIDefineByPos()` or `OCIDefineByPos2()`; for example, that define the attributes of an array fetch, `OCIDefineArrayOfStruct()`, or a named data type fetch, `OCIDefineObject()`. For example, to fetch multiple rows with a column of named data types, all the three calls must be invoked for the column. To fetch multiple rows of scalar columns only, `OCIDefineArrayOfStruct()` and `OCIDefineByPos()` or `OCIDefineByPos2()` are sufficient.

Oracle Database also provides predefined C data types that map object type attributes.

This section includes the following topics:

- About Defining LOB Output Variables
- About Defining PL/SQL Output Variables
- About Defining for a Piecewise Fetch

**See Also:**

- Object-Relational Data Types in OCI
- Advanced Define Operations in OCI
- OCIDefineByPos() or OCIDefineByPos2()
- OCIDefineArrayOfStruct()
- OCIDefineObject()

## 7.4.1 About Defining LOB Output Variables

There are two ways of defining LOBs:

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

This section includes the following topics:

- About Defining LOB Locators
- About Defining LOB Data

### 7.4.1.1 About 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, suppose that an application has prepared the following SQL statement:

```
SELECT lob1 FROM some_table;
```

In this statement, `lob1` is the LOB column, and `one_lob` is a define variable corresponding to a LOB column with the following declaration:

```
OCILobLocator * one_lob;
```

Then the following calls would be used to bind the placeholder and execute the statement:

```
/* initialize single locator */
OCIDescriptorAlloc(...&one_lob, OCI_DTYPE_LOB...);
...
/* pass the address of the locator */
OCIBindByName(...,(void *) &one_lob,... SQLT_CLOB, ...);
OCIStmtExecute(...,1,...);                /* 1 is the iters parameter */
```

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

```
OCILobLocator * lob_array[10];
...
for (i=0; i<10, i++)
   OCIDescriptorAlloc(...&lob_array[i], OCI_DTYPE_LOB...);
                                       /* initialize array of locators */
```

```
...
OCIBindByName(...,(void *) lob_array,...);
OCIStmtExecute(...,10,...);               /* 10 is the iters parameter */
```

Note that you must allocate descriptors with the `OCIDescriptorAlloc()` function before they can be used. In 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.

## 7.4.1.2 About Defining LOB Data

Oracle Database 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.

The following SQL statement is the basis for Example 7-14 and Example 7-15:

```
CREATE TABLE lob_tab (C1 CLOB, C2 CLOB);
```

**Example 7-14    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, (void *)buffer1, 8000,
                SQLT_LNG, (void *)0, (ub2 *)0, (ub2 *)0, (ub4) OCI_DEFAULT);
   OCIDefineByPos(stmthp, &defhp[1], errhp, 2, (void *)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);
}
```

**Example 7-15    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, (void *)buffer1, 8000,
                 SQLT_LNG, (void *)0, (ub2 *)0, (ub2 *)0, (ub4) OCI_DEFAULT);
   OCIDefineByPos(stmthp, &defhp[1], errhp, 2, (void *)buffer2, 8000,
                 SQLT_LNG, (void *)0, (ub2 *)0, (ub2 *)0, (ub4) OCI_DEFAULT);
   OCIStmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT);
}
```

### 7.4.2 About 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 Data Type and REF Defines, and PL/SQL OUT Binds for more information about defining PL/SQL output variables

### 7.4.3 About Defining for a Piecewise Fetch

A piecewise fetch requires an initial call to `OCIDefineByPos()` or `OCIDefineByPos2()`.

An additional call to `OCIDefineDynamic()` is necessary if the application uses callbacks rather than the standard polling mechanism.

**See Also:**

- OCIDefineByPos()or OCIDefineByPos2()
- OCIDefineDynamic()

## 7.5 About Binding and Defining Arrays of Structures in OCI

Defining arrays of structures requires an initial call to `OCIDefineByPos()` or `OCIDefineByPos2()`.

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 multirow, multicolumn 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.

To perform a multirow, multicolumn 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.

This section includes the following topics:

- Skip Parameters
- OCI Calls Used with Arrays of Structures
- Arrays of Structures and Indicator Variables

**See Also:**

- OCIDefineByPos() or OCIDefineByPos2()
- OCIDefineArrayOfStruct()
- OCIBindArrayOfStruct()

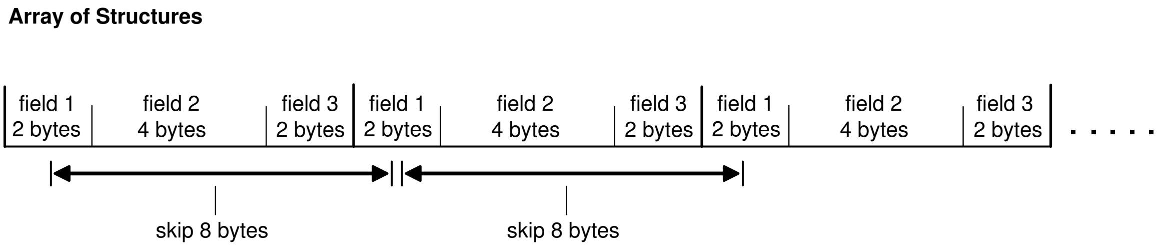
## 7.5.1 Skip Parameters

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 that you are binding or defining. This skip parameter is the number of bytes that must be skipped in the array of structures before the same field is encountered again. In general, this is equivalent to the byte size of one structure.

Figure 7-2 shows how a skip parameter is determined. In this case, the skip parameter is the sum of the sizes of the fields `field1` (2 bytes), `field2` (4 bytes), and `field3` (2 bytes), which is 8 bytes. This equals the size of one structure.

**Figure 7-2 Determining Skip Parameters**


Array of Structures
field 1
2 bytes
field 2
4 bytes
field 3
2 bytes
field 1
2 bytes
field 2
4 bytes
field 3
2 bytes
field 1
2 bytes
field 2
4 bytes
field 3
2 bytes
. . . . .
skip 8 bytes
skip 8 bytes

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 3 bytes of padding after the `ub1` so that the `ub4` that 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 produces a proper skip parameter of 8. On other systems, `skip_parameter` is set to 5 bytes by this statement. In the latter case, use the following statement to get the correct value for the skip parameter:

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

This section includes the following topic: Skip Parameters for Standard Arrays.

### 7.5.1.1 Skip Parameters for Standard Arrays

Arrays of structures are an extension of binding and defining arrays of single variables.

When you specify a single-variable array operation, the related skip equals the size of the data type of the array under consideration. For example, consider an array declared as follows:

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

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

## 7.5.2 OCI Calls Used with Arrays of Structures

What calls must be used when you perform operations involving arrays of structures.

Two OCI calls must be used when you perform operations involving arrays of structures:

- Use `OCIBindArrayOfStruct()` for binding fields in arrays of structures for input variables
- Use `OCIDefineArrayOfStruct()` for defining arrays of structures for output variables.

> **Note:**
>
> Binding or defining for arrays of structures requires multiple calls. A call to `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()` must precede a call to `OCIBindArrayOfStruct()`, and a call to `OCIDefineByPos()` or `OCIDefineByPos2()` must precede a call to `OCIDefineArrayOfStruct()`.

**See Also:**

- OCIBindArrayOfStruct()
- OCIDefineArrayOfStruct()
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIDefineByPos()or OCIDefineByPos2()

### 7.5.3 Arrays of Structures and Indicator Variables

The implementation of arrays of structures in addition 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
- OCIBindArrayOfStruct()
- OCIDefineArrayOfStruct()

## 7.6 About Binding and Defining Multiple Buffers

You can specify multiple buffers for use with a single bind or define call.

Performance is improved because the number of round-trips is decreased when data stored at different noncontiguous addresses is not copied to one contiguous location. CPU time spent and memory used are thus reduced.

The data type `OCIIOV` is defined as:

```
typedef struct OCIIOV
{
  void *bfp;  /* The pointer to a buffer for the data    */
  ub4  bfl;   /* The size of the buffer                  */
}OCIIOV;
```

The value `OCI_IOV` for the `mode` parameter is used in the `OCIBindByPos()` or `OCIBindByPos2()` and `OCIBindByName()` or `OCIBindByName2()` functions for binding multiple buffers. If this value of `mode` is specified, the address of `OCIIOV` must be passed in parameter `valuep`. The size of the data type must be passed in the parameter `valuesz`. For example:

```
OCIIOV vecarr[NumBuffers];
...
/* For bind at position 1 with data type int     */
OCIBindByPos(stmthp, bindp, errhp, 1, (void *)&vecarr[0],
             sizeof(int), ... OCI_IOV);
...
```

The value `OCI_IOV` for the mode parameter is used in the `OCIDefineByPos()` or `OCIDefineByPos2()` function for defining multiple buffers. If this value of `mode` is specified, the address of `OCIIOV` is passed in parameter `valuep`. The size of the data type must be passed in the parameter `valuesz`. This mode is intended to be used for scatter or gather binding, which allows multiple buffers to be bound or defined to a position, for example column A for the first 10 rows in one buffer, next 5 rows in one buffer, and the remaining 25 rows in another buffer. That eliminates the need to allocate and copy all of them into one big buffer while doing the array execute operation.

Example 7-16 illustrates the use of the structure `OCIIOV` and its `mode` values.

**Example 7-16    Using Multiple Bind and Define Buffers**

```
/* The following macros mention the maximum length of the data in the
 * different buffers. */

#define    LENGTH_DATE      10
#define    LENGTH_EMP_NAME  100


/* These two macros represent the number of elements in each bind and define
   array */
#define  NUM_BIND     30
#define  NUM_DEFINE   45

/* The bind buffers for inserting dates */
char  buf_1[NUM_BIND][LENGTH_DATE],
char  buf_2[NUM_BIND * 2][LENGTH_DATE],

/* The bind buffer for inserting emp name */
char  buf_3[NUM_BIND * 3][LENGTH_EMP_NAME],

/* The define buffers */
char  buf_4[NUM_DEFINE][LENGTH_EMP_NAME];
char  buf_5[NUM_DEFINE][LENGTH_EMP_NAME];

/* The size of data value for buffers corresponding to the same column must be
   the same, and that value is passed in the OCIBind or Define calls.
   buf_4 and buf_5 above have the same data values; that is, LENGTH_EMP_NAME
   although the  number of elements are different in the two buffers.

*/
OCIBind     *bndhp1 = (OCIBind   *)0;
OCIBind     *bndhp2 = (OCIBind   *)0;
OCIDefine   *defhp  = (OCIDefine *)0;
OCIStmt     *stmthp = (OCIStmt   *)0;
OCIError    *errhp  = (OCIError  *)0;

OCIIOV  bvec[2], dvec[2];

/*
Example of how to use indicators and return codes with this feature,
showing the allocation when using with define. You allocate memory
for indicator, return code, and the length buffer as one chunk of
NUM_DEFINE * 2 elements.
*/
short *indname[NUM_DEFINE*2];            /* indicators */
ub4   *alenname[NUM_DEFINE*2];           /* return lengths */
ub2   *rcodename[NUM_DEFINE*2];          /* return codes */

static text *insertstr  =
             "INSERT INTO EMP (EMP_NAME, JOIN_DATE) VALUES (:1, :2)";
static text *selectstr  = "SELECT EMP_NAME FROM EMP";

/* Allocate environment, error handles, and so on, and then initialize the
   environment.  */
...
/* Prepare the statement with the insert query in order to show the
   binds. */
OCIStmtPrepare  (stmthp, errhp, insertstr,
                 (ub4)strlen((char *)insertstr),
                 (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);

/* Populate buffers with values. The following represents the simplest
 * way of populating the buffers. However, in an actual scenario
 * these buffers may have been populated by data received from different
 * sources. */

/* Store the date in the bind buffers for the date. */
strcpy(buf_1[0], "21-SEP-02");
...
strcpy(buf_1[NUM_BIND - 1], "21-OCT-02");
...
strcpy(buf_2[0], "22-OCT-02");
...
strcpy(buf_2[2*NUM_BIND - 1], "21-DEC-02");
...
memset(bvec[0], 0, sizeof(OCIIOV));
memset(bvec[1], 0, sizeof(OCIIOV));

/* Set up the addresses in the IO Vector structure */
bvec[0].bfp = buf_1[0];                       /* Buffer address of the data */
bvec[0].bfl = NUM_BIND*LENGTH_DATE;           /* Size of the buffer */

/* And so on for other structures as well. */
bvec[1].bfp = buf_2[0];                       /* Buffer address of the data */
bvec[1].bfl = NUM_BIND*2*LENGTH_DATE;         /* Size of the buffer  */

/* Do the bind for date, using OCIIOV */
OCIBindByPos (stmthp, &bindhp2, errhp, 2, (void *)&bvec[0],
              sizeof(buf_1[0]), SQLT_STR,
              (void *)inddate, (ub2 *)alendate, (ub2 *)rcodedate, 0,
              (ub4 *)0,  OCI_IOV);
```

```
/* Store the employee names in the bind buffers, 3 for the names */
strcpy (buf_3[0], "JOHN ");
...
strcpy (buf_3[NUM_BIND *3 - 1], "HARRY");

/* Do the bind for employee name */
OCIBindByPos  (stmthp,  &bindhp1, errhp, 1, buf_3[0], sizeof(buf_3[0]),
      SQLT_STR, (void *)indemp, (ub2 *)alenemp, (ub2 *)rcodeemp, 0,
      (ub4 *)0, OCI_DEFAULT);

OCIStmtExecute (svchp, stmthp, errhp, NUM_BIND*3, 0,
               (OCISnapshot *)0, (OCISnapshot *)0, OCI_DEFAULT);

...
/* Now the statement to depict defines */
/* Prepare the statement with the select query in order to show the
   defines */
OCIStmtPrepare(stmthp, errhp, selectstr,(ub4)strlen((char *)selectstr),
               (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);

memset(dvec[0], 0, sizeof(OCIIOV);
memset(dvec[1], 0, sizeof(OCIIOV));

/* Set up the define vector */
dvec[0].bfp = buf_4[0];
dvec[0].bfl = NUM_DEFINE*LENGTH_EMP_NAME;

dvec[1].bfp = buf_5[0];
dvec[1].bfl = NUM_DEFINE*LENGTH_EMP_NAME;

/*
Pass the buffers for the indicator, length of the data, and the
return code.  Note that the buffer where you receive
the data is split into two locations,
each having NUM_DEFINE number of elements. However, the indicator
buffer, the actual length buffer, and the return code buffer comprise a
single chunk of NUM_DEFINE * 2 elements.
*/
OCIDefineByPos (stmthp, &defhp, errhp, 1, (void *)&dvec[0],
                sizeof(buf_4[0]), SQLT_STR, (void *)indname,
                (ub2 *)alenname, (ub2 *)rcodename, OCI_IOV);

OCIStmtExecute (svchp, stmthp, errhp, NUM_DEFINE*2, 0,
                (OCISnapshot*)0,
                (OCISnapshot*)0, OCI_DEFAULT);
...
```

**See Also:**

- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIDefineByPos() or OCIDefineByPos2()

# 7.7 DML with a RETURNING Clause in OCI

This section outlines the rules for correctly implementing DML statements with the `RETURNING` clause.

OCI supports the use of the `RETURNING` clause with SQL `INSERT`, `UPDATE`, and `DELETE` statements.

This section includes the following topics:

- About Using DML with a RETURNING Clause to Combine Two SQL Statements
- About Binding RETURNING...INTO Variables
- OCI Error Handling
- DML with RETURNING REF...INTO Clause in OCI
- Additional Notes About OCI Callbacks
- Array Interface for DML RETURNING Statements in OCI

> **See Also:**
>
> - The Database demonstration programs included with your Oracle installation for complete examples. For additional information, see OCI Demonstration Programs.
> - *Oracle Database SQL Language Reference* for more information about the use of the `RETURNING` clause with `INSERT`, `UPDATE`, or `DELETE` statements

## 7.7.1 About Using DML with a RETURNING Clause to Combine Two SQL Statements

Using the `RETURNING` clause with a DML statement enables you to combine two SQL statements into one, possibly saving 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 that 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
```

## 7.7.2 About 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 `OCIStmtExecute()` call, how much data is returned may not be known at run time.

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. By 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:

- Bind `RETURNING` clause placeholders in `OCI_DATA_AT_EXEC` mode using `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`, followed by a call to `OCIBindDynamic()` for each placeholder.
- 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.
- The `icbfp` parameter of `OCIBindDynamic()` call should provide a default function that returns `NULL` values when called.
- 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 is allowed between bind variables in the DML section and the `RETURNING` section of the statement.

**Note:**

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

**See Also:**

- OCIStmtExecute()
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIBindDynamic()

## 7.7.3 OCI Error Handling

The `OUT` bind function provided to `OCIBindDynamic()` must be prepared to receive partial results of a statement if there is an error.

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

## 7.7.4 DML with RETURNING REF...INTO Clause in OCI

The `RETURNING` clause can also be used to return a `REF` to an object that 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 postal code (ZIP)) in the `RETURNING` clause.

This section includes the following topic: Binding the Output Variable.

### 7.7.4.1 Binding the Output Variable

Binding the `REF` output variable in an OCI application requires three steps.

The following pseudocode in Example 7-17 shows a function that performs the binds necessary for the preceding three steps.

1. Set the initial bind information is set using `OCIBindByName()` or `OCIBindByName2()`.
2. Set additional bind information for the `REF`, including the type description object (TDO), is set with `OCIBindObject()`.
3. Make a call is made to `OCIBindDynamic()`.

### Example 7-17    Binding the REF Output Variable in an OCI Application

```
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"),
                       (void *) 0, (sb4) sizeof(OCIRef *), SQLT_REF,
                       (void *) 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"),
                       (void *) 0, (sb4) MAXZIPLEN, SQLT_CHR,
                       (void *) 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,
          (void **) &addrref[0], (ub4 *) 0, (void **) 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, (void *) &pos[0], cbf_no_data,
                    (void *) &pos[0], cbf_get_data)
  ||  OCIBindDynamic(bndhp[3], errhp, (void *) &pos[1], cbf_no_data,
                    (void *) &pos[1], cbf_get_data))
  {
    return OCI_ERROR;
  }

  return OCI_SUCCESS;
}
```

**See Also:**

- OCIBindByName() or OCIBindByName2()
- OCIBindObject()
- OCIBindDynamic()

### 7.7.5 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 increment the buffer pointer to the correct memory within the allocated space.

### 7.7.6 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 iterations 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.

**See Also:**

OCIStmtExecute()

## 7.8 Character Conversion in OCI Binding and Defining

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

This section includes the following topics:

- About Choosing a Character Set
- About Setting Client Character Sets in OCI
- About Binding Variables in OCI

## 7.8.1 About Choosing a Character Set

If a database column containing character data is defined to be a `CHAR` or `VARCHAR2` or `NCHAR` or `NVARCHAR2` column, then a bind or define involving that column must make 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 up to a factor of four. Ensure that buffers that are provided to hold the data are of sufficient size.

Beginning with Oracle Database 12*c* Release 2 (12.2), OCI provides two service context handle attributes `OCI_ATTR_MAX_CHARSET_RATIO` and `OCI_ATTR_MAX_NCHARSET_RATIO` to obtain the maximum character set expansion ratio from server to client character set or national character set respectively. Using these attributes lets you more efficiently allocate optimal memory of buffers before conversion so that when data is returned from the database, sufficient space can be allocated to hold it. Using these attributes is useful in scenarios where there are different character sets or national character sets between server and client.

The following code example shows how get the maximum character set expansion ratio from server to client character set:

```
size_t   cratio;
OCIAttrGet((void *)svchp, (ub4)OCI_HTYPE_SVCCTX, (size_t *)&cratio, (ub4)
0, OCI_ATTR_MAX_CHARSET_RATIO, errhp);
printf("Conversion ratio from server to client character set is %d\n",
cratio);

Conversion ratio from server to client character set is 2
```

The following code example shows how get the maximum character set expansion ratio from server to client national character set:

```
size_t   cratio;
OCIAttrGet((void *)svchp, (ub4)OCI_HTYPE_SVCCTX, (size_t *)&cratio, (ub4)
0, OCI_ATTR_MAX_NCHARSET_RATIO, errhp);
printf("Conversion ratio from server to client ncharset is %d\n", cratio);

Conversion ratio from server to client ncharset is 1
```

In some cases, it may also be easier for an application to deal with `CHAR` or `VARCHAR2` or with `NCHAR` or `NVARCHAR2` data in terms of numbers of characters, rather than numbers of bytes, which is the usual case.

This section includes the following topics:

- Character Set Form and ID
- Implicit Conversion Between CHAR and NCHAR

### 7.8.1.1 Character Set Form and ID

Each OCI bind and define handle is associated with the `OCI_ATTR_CHARSET_FORM` and `OCI_ATTR_CHARSET_ID` attributes.

An application can set these attributes with the `OCIAttrSet()` call to specify the character form and character set ID of the bind or 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 that the 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 SQL Language Reference* for more information about `NCHAR` data
- OCIAttrSet()

### 7.8.1.2 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`.

Although 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.

## 7.8.2 About Setting Client Character Sets in OCI

You can set the client 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. The `ncharset` parameter controls coding of `NCHAR` data. The function `OCINlsEnvironmentVariableGet()` returns the character set from `NLS_LANG` and the national character set from `NLS_NCHAR`.

Example 7-18 illustrates the use of these functions (OCI provides a typedef called `utext` to facilitate binding and defining of UTF-16 data):

**See Also:**

- OCIEnvNlsCreate()
- OCINlsEnvironmentVariableGet()

**Example 7-18 Setting the Client Character Set to OCI_UTF16ID in OCI**

```
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*/
...
```

## 7.8.3 About Binding Variables in OCI

Update or insert operations are done through variable binding.

When binding variables, specify the `OCI_ATTR_MAXDATA_SIZE` attribute and `OCI_ATTR_MAXCHAR_SIZE` attribute in the bind handle to indicate the byte and character constraints used when inserting data in to the Oracle database.

These attributes are defined as:

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

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

This section includes these additional topics:

- Buffer Expansion During OCI Binding
- Constraint Checking During Defining
- General Compatibility Issues for Character-Length Semantics in OCI

> **See Also:**
>
> - About Using the OCI_ATTR_MAXDATA_SIZE Attribute for more information
> - About Using the OCI_ATTR_MAXCHAR_SIZE Attribute for more information

## 7.8.3.1 About Using the OCI_ATTR_MAXDATA_SIZE Attribute

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 typically sets `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 Database issues an error if `OCI_ATTR_MAXDATA_SIZE` is not large enough to accommodate the data after conversion, and the operation fails.

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 uses of the `OCI_ATTR_MAXDATA_SIZE` attribute:

- Scenario 1: `CHAR` (source data) converted to 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 Oracle Database character sets.

- Scenario 2: `CHAR` (source data) converted to `CHAR` (destination column) or non-`CHAR` (source data) converted to `CHAR` (destination column)

  The recommended value of `OCI_ATTR_MAXDATA_SIZE` is the size of the column.

- Scenario 3: CHAR (source data) converted to a PL/SQL variable

  In this case, the recommended value of `OCI_ATTR_MAXDATA_SIZE` is the size of the PL/SQL variable.

### 7.8.3.2 About Using the 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 in the Oracle database 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 in the Oracle database 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()` or `OCIDefineByPos2()` 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.

**See Also:**

OCIDefineByPos() or OCIDefineByPos2()

### 7.8.3.3 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.

This section includes the following topics:

- IN Binds
- Dynamic SQL
- Buffer Expansion During Inserts

#### 7.8.3.3.1 IN Binds

For an `IN` bind, 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 fewer 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.

#### 7.8.3.3.2 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.

#### 7.8.3.3.3 Buffer Expansion During Inserts

Use `OCI_ATTR_MAXDATA_SIZE` to 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 `OCIBindByPos2()` or `OCIBindByName()` or `OCIBindByName2()` 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 fits 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 returns an error. To avoid this problem specify the `OCI_ATTR_MAXDATA_SIZE` to be 4000 to guarantee that the Oracle database never exceeds 4000 bytes.

**See Also:**

- OCIBindByPos() or OCIBindByPos2()
- OCIBindByName() or OCIBindByName2()

### 7.8.3.4 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 because the buffer size in bytes serves as the limit on byte length. The define buffer size provided in the `OCIDefineByPos()` or `OCIDefineByPos2()` call can be used as the byte constraint.

This section includes the following topics:

- Dynamic SQL Selects
- Return Lengths

**See Also:**

OCIDefineByPos() or OCIDefineByPos2()

### 7.8.3.4.1 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 the `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.

### 7.8.3.4.2 Return Lengths

The following return 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 data types such as `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 the `OCI_UTF16ID` character set ID; then the return lengths are in UTF-16 units.

**Note:**

The buffer sizes in the bind and define calls and the piece sizes in the `OCIStmtGetPieceInfo()` and `OCIStmtSetPieceInfo()` and the callbacks are always in bytes.

**See Also:**

- OCIStmtGetPieceInfo()
- OCIStmtSetPieceInfo()

## 7.8.3.5 General Compatibility Issues for Character-Length Semantics in OCI

Character-length semantics in OCI depends on the Oracle Database release, release 9.0 or later versus release 8.1 or earlier.

- For a release 9.0 or later client communicating with a release 8.1 or earlier Oracle Database, `OCI_ATTR_MAXCHAR_SIZE` is not known by the Oracle Database, so this value is ignored. If you specify only this value, OCI derives the corresponding

`OCI_ATTR_MAXDATA_SIZE` value based on the maximum number of bytes for each character for the client-side character set.

- For a release 8.1 or earlier client communicating with a release 9.0 or later Oracle Database, the client can never specify an `OCI_ATTR_MAXCHAR_SIZE` value, so the Oracle Database considers the client as 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 Oracle database and client can exchange information in an appropriate manner.

This section includes the following topics:

- Code Example for Inserting and Selecting Using OCI_ATTR_MAXCHAR_SIZE
- Code Example for UTF-16 Binding and Defining

### 7.8.3.5.1 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 database considers the worst case scenario.

This is shown in Example 7-19. The real number of bytes allocated is a multiple of `N`, say `M` times `N`. Currently, `M` is 3 as the maximum number of bytes allocated for each character in UTF-8.

For example, in Example 7-19, in the `EMP` table, the `ENAME` column is defined as 30 characters and the `ADDRESS` column is defined as 80 characters. Thus, the corresponding byte lengths in the database are M*30 or 3*30=90, and M*80 or 3*80=240, respectively.

**Example 7-19 Insert and Select Operations Using the OCI_ATTR_MAXCHAR_SIZE Attribute**

```
...
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"),
    (void *)ename, sizeof(ename), SQLT_STR, (void *)&insname_ind,
    (ub2 *)alenp, (ub2 *)rcodep, (ub4)maxarr_len, (ub4 *)curelep, OCI_DEFAULT);
/* either */
OCIAttrSet((void *)bnd1p, (ub4)OCI_HTYPE_BIND, (void *)&ename_max_bytes,
    (ub4)0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
/* or */
OCIAttrSet((void *)bnd1p, (ub4)OCI_HTYPE_BIND, (void *)&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, (void *)ename,
   (sb4)sizeof (ename),
   SQLT_STR, (void *)&selname_ind, (ub2 *)alenp, (ub2 *)rcodep,
   (ub4)OCI_DEFAULT);
/* if not called, byte semantics is by default */
OCIAttrSet((void *)dfn1p, (ub4)OCI_HTYPE_DEFINE, (void *)&ename_max_chars,
   (ub4)0,
   (ub4)OCI_ATTR_MAXCHAR_SIZE, errhp);
...
```

### 7.8.3.5.2 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 `NVARCHAR2` variant handles can be set to assume that all data is passed in UTF-16 (Unicode) encoding. To specify UTF-16, set `OCI_ATTR_CHARSET_ID` = `OCI_UTF16ID`.

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` data type conforms to UTF-16 can easily convert `utext` to the `wchar_t` data type 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` data type as the buffer for input and output data.

Example 7-20 shows pseudocode that illustrates a bind and define for UTF-16 data.

**Example 7-20    Binding and Defining 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"),
              (void *) ename, sizeof(ename), SQLT_STR,
              (void *)&insname_ind, (ub2 *) 0, (ub2 *) 0, (ub4) 0,
              (ub4 *)0, OCI_DEFAULT);
OCIAttrSet ((void *) bnd1p, (ub4) OCI_HTYPE_BIND, (void *) &csid,
           (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAttrSet((void *) bnd1p, (ub4) OCI_HTYPE_BIND, (void *) &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, (void *)ename,
                (sb4)sizeof(ename), SQLT_STR,
                (void *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);
OCIAttrSet ((void *) dfn1p, (ub4) OCI_HTYPE_DEFINE, (void *) &csid,
            (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
...
```

**See Also:**

Bind Handle Attributes

# 7.9 PL/SQL REF CURSORs and Nested Tables in OCI

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`.

In this code in Example 7-21, `stm1p` is the statement handle for the PL/SQL block, whereas `stm2p` is the statement handle that 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 shown in Example 7-22.

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, after 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`.

**Example 7-21 Binding the :cursor1 Placeholder to the Statement Handle stm2p as a REF CURSOR**

```
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"),
             (void *)&stm2p, (sb4) 0,  SQLT_RSET, (void *)0,
               (ub2 *)0, (ub2 *)0, (ub4)0, (ub4 *)0,   (ub4)OCI_DEFAULT);
```

**Example 7-22 Defining a Nested Table (Second Position) as a Statement Handle**

```
status = OCIStmtPrepare (stm1p, errhp, (text *) nst_tab,
         strlen((char *)nst_tab), OCI_NTV_SYNTAX, OCI_DEFAULT);
...
status = OCIDefineByPos (stm1p, (OCIDefine **) &dfn2p, errhp, (ub4)2,
          (void *)&stm2p, (sb4)0, SQLT_RSET, (void *)0, (ub2 *)0,
                     (ub2 *)0, (ub4)OCI_DEFAULT);
```

**See Also:**

OCIHandleAlloc()

# 7.10 Natively Describe and Bind All PL/SQL Types Including Package Types

Beginning with Oracle Database Release 12.1, OCI clients support the ability to natively describe and bind all PL/SQL types.

This includes the base scalar type Boolean, which was previously unsupported as a bind type. This also includes types declared in PL/SQL packages, such as named record or collection type (including nested table, varray and index table) or implicit record subtype (%rowtype) declared inside of a PL/SQL package specification. Native support for these features means clients can describe and bind PL/SQL types using only the provided client-side APIs.

The PL/SQL typecodes for these data types (Boolean, record, index-by `BINARY_INTEGER`, and `PLS_INTEGER` or `BINARY_INTEGER`) are listed in Table 5-10. The equivalent `SQLT` type for these PL/SQL typecodes is listed in Table 5-11. Clients must bind the specified type using the respective specified value of `SQLT` type as the DTY of the bind. For example, for records, clients must bind package record types (`OCI_TYPECODE_RECORD`) using `SQLT_NTY` as the DTY of the bind; for collections, clients must bind all package collection types (`OCI_TYPECODE_ITABLE`) using `SQLT_NTY` as the DTY of the bind; and for Booleans, clients must bind Boolean types (`OCI_TYPECODE_BOOLEAN`) using `SQLT_BOL` as the DTY of the bind. Bind APIs: `OCIBindByName()`, `OCIBindByName2()`, `OCIBindByPos()`, and `OCIBindByPos2()` support each `SQLT` type value in the DTY of the bind that represents these PL/SQL typecodes.

> **See Also:**
>
> - OCIBindByName() or OCIBindByName2()
> - OCIBindByPos() or OCIBindByPos2()

## 7.11 Runtime Data Allocation and Piecewise Operations in OCI

You can use OCI to perform piecewise inserts, updates, and fetches of data.

You can also use 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 run time 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 `OCIStmtFetch2()` 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 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 `OCIStmtFetch2()` once more than the number of pieces to be fetched.

This section includes the following topics:

- Valid Data Types for Piecewise Operations
- Types of Piecewise Operations

- About Providing INSERT or UPDATE Data at Runtime
- Piecewise Operations with PL/SQL
- PL/SQL Indexed Table Binding Support
- About Providing FETCH Information at Run Time
- Piecewise Binds and Defines for LOBs

> **See Also:**
>
> - OCIStmtFetch2()
> - OCIStmtExecute()

## 7.11.1 Valid Data Types for Piecewise Operations

Only some data types can be manipulated in pieces.

OCI applications can perform piecewise fetches, inserts, or updates of all the following data types:

- `VARCHAR2`
- `STRING`
- `LONG`
- `LONG RAW`
- `RAW`
- `CLOB`
- `BLOB`

Another way of using this feature for all data types 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 data types that do not support piecewise operations.

## 7.11.2 Types of Piecewise Operations

What are the ways you can perform 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 `OCIBindByPos2()` or `OCIBindByName()` or `OCIBindByName2()` call to `OCI_DATA_AT_EXEC`, it indicates that an OCI application is providing data for an `INSERT` or `UPDATE` operation dynamically at runtime.

Similarly, when you set the `mode` parameter of an `OCIDefineByPos()` or `OCIDefineByPos2()` call to `OCI_DYNAMIC_FETCH`, it indicates that an application dynamically provides allocation space for receiving data at the time of the fetch.

In each case, you can provide the runtime information for the `INSERT`, `UPDATE`, or `FETCH` operation 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 runtime data allocation and piecewise operations for inserts, updates, and fetches.

**Note:**

Piecewise operations are also valid for SQL and PL/SQL blocks.

**See Also:**

- OCIBindByPos() or OCIBindByPos2()
- OCIBindByName() or OCIBindByName2()
- OCIDefineByPos() or OCIDefineByPos2()

## 7.11.3 About Providing INSERT or UPDATE Data at Runtime

When you specify the `OCI_DATA_AT_EXEC` mode in a call to `OCIBindByPos()` or `OCIBindByPos2()` or `OCIBindByName()` or `OCIBindByName2()`, the `value_sz` parameter defines the total size of the data that can be provided at run time.

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 two ways:

- You can define a callback using the `OCIBindDynamic()` function, which when called at run time returns either a piece of the data or all of it.
- 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.

This section includes the following topic: Performing a Piecewise Insert or Update.

**See Also:**

- OCIBindByPos() or OCIBindByPos2()
- OCIBindByName() or OCIBindByName2()
- OCIBindDynamic()
- OCIStmtExecute()
- OCIStmtSetPieceInfo()

### 7.11.3.1 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 data types.

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 must be inserted, populates a buffer with that piece, and then executes an insert. When the last piece has been inserted, the operation is complete.

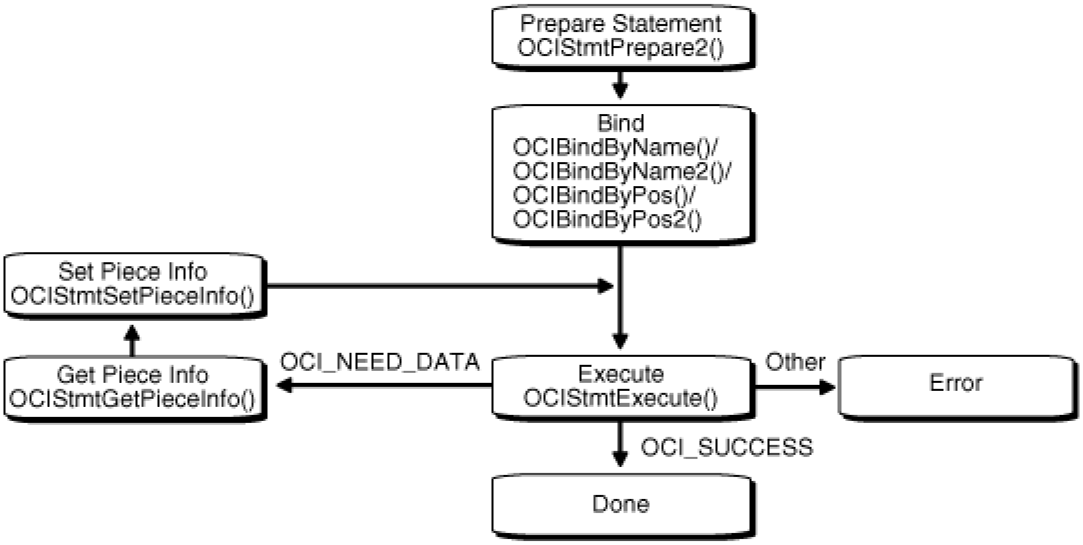
Keep in mind that the insert buffer can be of arbitrary size and is provided at run time. In addition, each inserted piece does not need to be of the same size. The size of each piece to be inserted is established by each `OCIStmtSetPieceInfo()` call.

**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 is expected to be smaller. You must account for this by indicating the smaller size in the final `OCIStmtSetPieceInfo()` call.

The procedure is illustrated in Figure 7-3 and expanded in the steps following the figure.

**Figure 7-3 Performing Piecewise Insert**


Prepare Statement
OCIStmtPrepare2()
Bind
OCIBindByName()/
OCIBindByName2()/
OCIBindByPos()/
OCIBindByPos2()
Set Piece Info
OCIStmtSetPieceInfo()
Get Piece Info
OCIStmtGetPieceInfo()
OCI_NEED_DATA
Execute
OCIStmtExecute()
Other
Error
OCI_SUCCESS
Done

1. Initialize the OCI environment, allocate the necessary handles, connect to a server, authorize a user, and prepare a statement request by using `OCIStmtPrepare2()`.
2. Bind a placeholder by using `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`. You do not need to specify the actual size of the pieces you use, but you must provide the total size of the data that can be provided at run time.
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 must 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 first piece (`OCI_FIRST_PIECE`), an intermediate piece (`OCI_NEXT_PIECE`), or 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.

> **See Also:**
>
> - Polling Mode Operations in OCI
> - OCIBindDynamic()
> - OCIStmtExecute()
> - OCIStmtGetPieceInfo()
> - OCIStmtSetPieceInfo()
> - OCIStmtPrepare2()
> - OCIBindByName() or OCIBindByName2()
> - OCIBindByPos() or OCIBindByPos2()

## 7.11.4 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.

> **See Also:**
>
> OCIBindDynamic()

## 7.11.5 PL/SQL Indexed Table Binding Support

PL/SQL indexed tables can be passed as `IN/OUT` binds into PL/SQL anonymous blocks using OCI.

The procedure for binding PL/SQL indexed tables is quite similar to performing an array bind for SQL statements. The OCI program must bind the location of an array with other metadata for the array as follows, using either `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`. The process of binding a C array into a PL/SQL indexed table bind variable must provide the following information during the bind call:

- `void *valuep (IN/OUT)` - A pointer to a location that specifies the beginning of the array in client memory
- `ub2 dty (IN)` - The data type of the elements of the array as represented on the client

- `sb4 value_sz (IN)` - The maximum size (in bytes) of each element of the array as represented on the client
- `ub4 maxarr_len (IN)` - The maximum number of elements of the data type the array is expected to hold in its lifetime

  If allocating the entire array up front for doing static bindings, the array must be sized sufficiently to contain `maxarr_len` number of elements, each of size `value_sz`. This information is also used to constrain the indexed table as seen by PL/SQL. PL/SQL cannot look up the indexed table (either for read or write) beyond this specified limit.
- `ub4 *curelep (IN/OUT)` - A pointer to the number of elements in the array (from the beginning of the array) that are currently valid.

  This should be less than or equal to the maximum array length. Note that this information is also used to constrain the indexed table as seen by PL/SQL. For `IN` binds, PL/SQL cannot read from the indexed table beyond this specified limit. For `OUT` binds, PL/SQL can write to the indexed table beyond this limit, but not beyond the `maxarr_len` limit.

For `IN` indexed table binds, before performing `OCIStmtExecute()`, the user must set up the current array length (`*curelep`) for that execution. In addition, the user also must set up the actual length and indicator as applicable for each element of the array.

For `OUT` binds, OCI must return the current array length (`*curelep`) and the actual length, indicator and return code as applicable for each element of the array.

For best performance, keep the array allocated with maximum array length, and then vary the current array length between executes based on how many elements are actually being passed back and forth. Such an approach does not require repeatedly deallocating and reallocating the array for every execute, thereby helping overall application performance.

It is also possible to bind using OCI piecewise calls for PL/SQL indexed tables. Such an approach does not require preallocating the entire array up front. The `OCIStmtSetPieceInfo()` and `OCIStmtGetPieceInfo()` calls can be used to pass in individual elements piecewise.

This section includes the following topic: Restrictions for PL/SQL Indexed Table Binding Interface.

**See Also:**

- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIStmtExecute()
- OCIStmtSetPieceInfo()
- OCIStmtGetPieceInfo()

### 7.11.5.1 Restrictions for PL/SQL Indexed Table Binding Interface

What are the restrictions for the PL/SQL indexed table OCI binding interface.

The PL/SQL indexed table OCI binding interface does not support binding:

- Arrays of ADTs or `REF`s
- Arrays of descriptor types such as LOB descriptors, ROWID descriptors, datetime or interval descriptors
- Arrays of PLSQL record types

## 7.11.6 About Providing FETCH Information at Run Time

When a call is made to `OCIDefineByPos()` or `OCIDefineByPos2()` 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 a callback function that is invoked to get information about your data buffer.

Runtime data is provided in one of two ways:

- You can define a callback using the `OCIDefineDynamic()` function. The `value_sz` parameter defines the maximum size of the data that is provided at run time. When the client library needs a buffer to return the fetched data, the callback is invoked to provide a runtime buffer into which either a piece of the data or all of it is 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 by using `OCIStmtSetPieceInfo()`. The `OCIStmtGetPieceInfo()` call provides information about which define and which piece are involved.

This section includes the following topic: Performing a Piecewise Fetch

**See Also:**

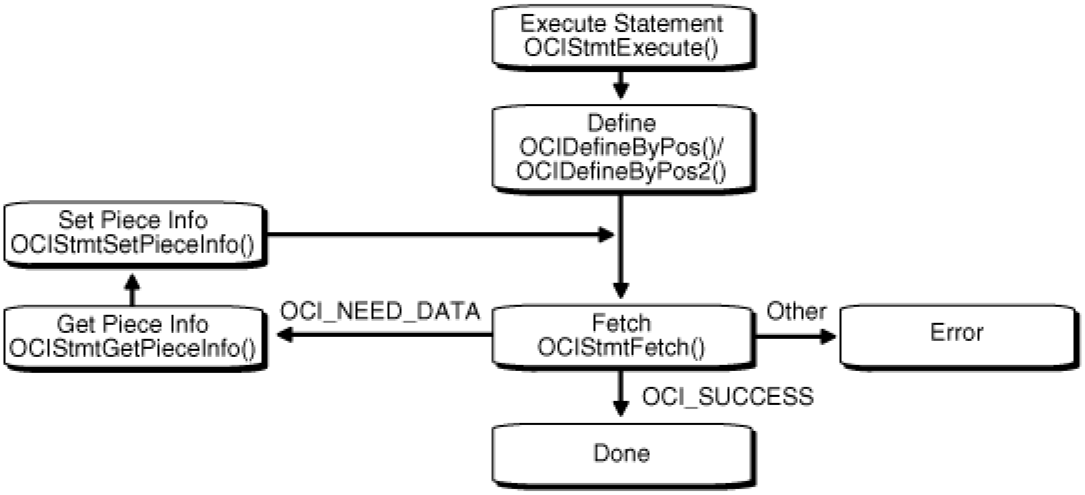
- OCIDefineByPos() or OCIDefineByPos2()
- OCIDefineDynamic()
- OCIStmtSetPieceInfo()
- OCIStmtGetPieceInfo()

### 7.11.6.1 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 7-4 and explained in the steps following the figure.

**Figure 7-4 Performing Piecewise Fetch**


Execute Statement
OCIStmtExecute()
Define
OCIDefineByPos()/
OCIDefineByPos2()
Set Piece Info
OCIStmtSetPieceInfo()
Get Piece Info
OCIStmtGetPieceInfo()
OCI_NEED_DATA
Fetch
OCIStmtFetch()
Other
Error
OCI_SUCCESS
Done

1. Initialize the OCI environment, allocate necessary handles, connect to a database, authorize a user, prepare a statement, and execute the statement by using `OCIStmtExecute()`.
2. Define an output variable by using `OCIDefineByPos()` or `OCIDefineByPos2()`, with `mode` set to `OCI_DYNAMIC_FETCH`. At this point you do not need to specify the actual size of the pieces you use, but you must provide the total size of the data that is to be fetched at run time.
3. Call `OCIStmtFetch2()` 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, then 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 `OCIStmtFetch2()` again to retrieve the actual piece. If `OCIStmtFetch2()` returns `OCI_SUCCESS`, all the pieces have been fetched successfully. If `OCIStmtFetch2()` returns `OCI_NEED_DATA`, return to Step 4 to process the next piece. If any other value is returned, an error occurred.

**See Also:**

- Polling Mode Operations in OCI
- OCIStmtSetPieceInfo()
- OCIStmtExecute()
- OCIDefineByPos() or OCIDefineByPos2()
- OCIStmtGetPieceInfo()
- OCIStmtFetch2()

## 7.11.7 Piecewise Binds and Defines for LOBs

What are the ways of doing piecewise binds and defines for LOBs.

There are two:

- 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 data type 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 data type for these functions:

  - `OCIDefineByPos()` or `OCIDefineByPos2()`
  - `OCIBindByName()` or `OCIBindByName2()`
  - `OCIBindByPos()` or `OCIBindByPos2()`

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

- 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 data type for the following functions.

  - `OCIDefineByPos()` or `OCIDefineByPos2()`
  - `OCIBindByName()` or `OCIBindByName2()`
  - `OCIBindByPos()` or `OCIBindByPos2()`

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

**See Also:**

- OCIDefineByPos() or OCIDefineByPos2()
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- About Binding LOB Data for usage and examples for both `INSERT` and `UPDATE statements`
- About Defining LOB Data for usage and examples of `SELECT` statements
- OCILobRead2()
- OCILobWrite2()
- LOB Read and Write Callbacks for information about streaming using callbacks with `OCILobWrite2()` and `OCILobRead2()`

# 8

# Describing Schema Metadata

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

This chapter contains these topics:

- About Using OCIDescribeAny()
- Parameter Attributes
- Character-Length Semantics Support in Describe Operations
- Examples Using OCIDescribeAny()

> **See Also:**
>
> OCIDescribeAny()

## 8.1 About Using OCIDescribeAny()

Performs an explicit describe of schema objects and their subschema objects.

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

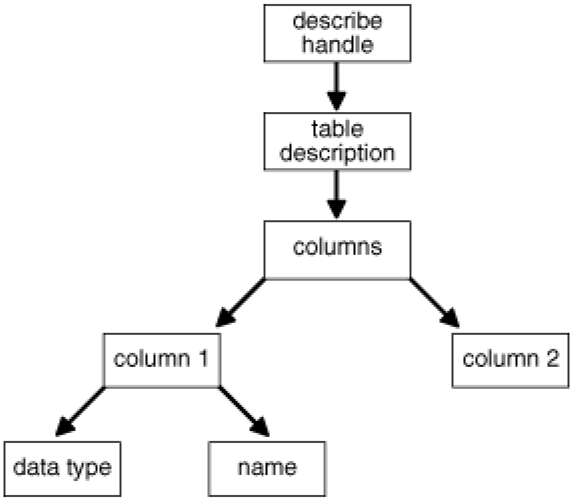
Information about other schema elements (function arguments, columns, type attributes, and type methods) is available through a describe of one of the preceding schema objects or an explicit describe of the subschema object.

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 8-1.

**Figure 8-1 OCIDescribeAny() Table Description**



The describe handle returned by the `OCIDescribeAny()` call has an attribute, `OCI_ATTR_PARAM`, that points to such a description tree. Each node of the tree has attributes associated with that node, as well as attributes that are like recursive describe handles and point to subtrees containing further information. If all the attributes are homogenous, as with 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 data type.

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 the entire description is cached on the client side by `OCIDescribeAny()`.

This section includes the following topics:

- Limitations on OCIDescribeAny()
- Notes on Types and Attributes

**See Also:**

- OCIDescribeAny()
- OCIHandleAlloc()
- OCIAttrGet()
- OCIParamGet()

## 8.1.1 Limitations on OCIDescribeAny()

The `OCIDescribeAny()` call limits information returned to the basic information and stops expanding a node if it amounts to another describe operation.

For example, if a table column is of an object type, then OCI does not return a subtree describing the type, because this information can be obtained by another describe call.

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:**

- About Describing Select-List Items
- OCIDescribeAny()
- OCIStmtExecute()

## 8.1.2 Notes on Types and Attributes

What to be aware of when performing describe operations.

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

- Data Type Codes
- About Describing Types
- Implicit and Explicit Describe Operations
- OCI_ATTR_LIST_ARGUMENTS Attribute

### 8.1.2.1 Data Type Codes

The `OCI_ATTR_TYPECODE` attribute returns typecodes that 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 type (boolean) is supported.

The `OCI_ATTR_DATA_TYPE` attribute returns typecodes that represent the data types stored in database columns. These are similar to the describe values returned by previous versions of Oracle Database. These values are represented by SQLT constants (`ub2` values). Boolean types return `SQLT_BOL`.

> **See Also:**
>
> - External Data Types for more information about `SQLT_BOL`
> - Typecodes for more information about typecodes, such as the `OCI_TYPECODE` values returned in the `OCI_ATTR_TYPECODE` attribute and the SQLT typecodes returned in the `OCI_ATTR_DATA_TYPE` attribute

### 8.1.2.2 About Describing Types

To describe type objects, it is necessary to initialize the OCI process in object mode.

This is shown in Example 8-1.

**Example 8-1 Initializing the OCI Process in Object Mode**

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

> **See Also:**
>
> OCIEnvCreate()

### 8.1.2.3 Implicit and Explicit Describe Operations

The column attribute `OCI_ATTR_PRECISION` can be returned using an implicit describe with `OCIStmtExecute()` and an explicit describe with `OCIDescribeAny()`.

When you use an implicit describe, set the precision to `sb2`. When you use an explicit describe, set the precision to `ub1` for a placeholder. This is necessary to match the data type of precision in the dictionary.

**See Also:**

- OCIStmtExecute()
- OCIDescribeAny()

### 8.1.2.4 OCI_ATTR_LIST_ARGUMENTS Attribute

The `OCI_ATTR_LIST_ARGUMENTS` attribute for type methods represents *second-level* arguments for the method.

For example, consider the following record `my_type` and the procedure `my_proc` that takes an argument of type `my_type`:

```
my_type record(a number, b char)
my_proc (my_input my_type)
```

In this example, the `OCI_ATTR_LIST_ARGUMENTS` attribute would apply to arguments `a` and `b` of the `my_type` record.

## 8.2 Parameter Attributes

This section describes the attributes and handles that belong to different parameters.

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.

The `OCIDescribeAny()` call does support more than two name components (for example, `schema.type.attr1.attr2.method1`). 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 8-1 lists the attributes of all parameters.

**Table 8-1 Attributes of All Parameters**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `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 8-1 (Cont.) Attributes of All Parameters**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_PTYPE` | Type of information described by the parameter. Possible values:<br>`OCI_PTYPE_TABLE` - table<br>`OCI_PTYPE_VIEW` - view<br>`OCI_PTYPE_PROC` - procedure<br>`OCI_PTYPE_FUNC` - function<br>`OCI_PTYPE_PKG` - package<br>`OCI_PTYPE_TYPE` - type, including a package type<br>`OCI_PTYPE_TYPE_ATTR` - attribute of a type, including package record type attributes<br>`OCI_PTYPE_TYPE_COLL` - collection type information, including package collection elements<br>`OCI_PTYPE_TYPE_METHOD` - method of a type<br>`OCI_PTYPE_SYN` - synonym<br>`OCI_PTYPE_SEQ` - sequence<br>`OCI_PTYPE_COL` - column of a table or view<br>`OCI_PTYPE_ARG` - argument of a function or procedure<br>`OCI_PTYPE_TYPE_ARG` - argument of a type method<br>`OCI_PTYPE_TYPE_RESULT` - results of a method<br>`OCI_PTYPE_LIST` - column list for tables and views, argument list for functions and procedures, or subprogram list for packages<br>`OCI_PTYPE_SCHEMA` - schema<br>`OCI_PTYPE_DATABASE` - database<br>`OCI_PTYPE_UNK` - unknown schema object | `ub1` |
| `OCI_ATTR_TIMESTAMP` | The time stamp 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
- Procedure, Function, and Subprogram Attributes
- Package Attributes
- Type Attributes
- Type Attribute Attributes
- Type Method Attributes
- Collection Attributes
- Synonym Attributes
- Sequence Attributes
- Column Attributes

- Argument and Result Attributes
- List Attributes
- Schema Attributes
- Database Attributes
- Rule Attributes
- Rule Set Attributes
- Evaluation Context Attributes
- Table Alias Attributes
- Variable Type Attributes
- Name Value Attributes

> **See Also:**
>
> - OCIParamGet()
> - OCIDescribeAny()

## 8.2.1 Table or View Parameters

Lists and describes the type-specific attributes for parameters for a table or view.

Table 8-2 lists the type-specific attributes for parameters for a table or view (type `OCI_PTYPE_TABLE` or `OCI_PTYPE_VIEW`).

**Table 8-2 Attributes of Tables or Views**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_OBJID` | Object ID | `ub4` |
| `OCI_ATTR_NUM_COLS` | Number of columns | `ub2` |
| `OCI_ATTR_LIST_COLUMNS` | Column list (type `OCI_PTYPE_LIST`) | `OCIParam *` |
| `OCI_ATTR_REF_TDO` | REF to the type description object (TDO) of the base type for extent tables | `OCIRef *` |
| `OCI_ATTR_IS_TEMPORARY` | Indicates that the table is temporary | `ub1` |
| `OCI_ATTR_IS_TYPED` | Indicates that the table is typed | `ub1` |
| `OCI_ATTR_DURATION` | Duration of a temporary table. Values can be:<br>`OCI_DURATION_SESSION` - session<br>`OCI_DURATION_TRANS` - transaction<br>`OCI_DURATION_NULL` - table not temporary | `OCIDuration` |

Table 8-3 lists additional attributes that belong to tables.

**Table 8-3    Attributes Specific to Tables**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_RDBA` | Data block address of the segment header | `ub4` |
| `OCI_ATTR_TABLESPACE` | Tablespace that the table resides in | `word` |
| `OCI_ATTR_CLUSTERED` | Indicates that the table is clustered | `ub1` |
| `OCI_ATTR_PARTITIONED` | Indicates that the table is partitioned | `ub1` |
| `OCI_ATTR_INDEX_ONLY` | Indicates that the table is index-only | `ub1` |

## 8.2.2 Procedure, Function, and Subprogram Attributes

Lists and describes the type-specific attributes when a parameter is for a procedure or function.

Table 8-4 lists the type-specific attributes when a parameter is for a procedure or function (type `OCI_PTYPE_PROC` or `OCI_PTYPE_FUNC`).

**Table 8-4    Attributes of Procedures or Functions**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_LIST_ARGUMENTS` | Argument list. See List Attributes. | `void *` |
| `OCI_ATTR_IS_INVOKER_RIGHTS` | Indicates that the procedure or function has invoker's rights | `ub1` |

Table 8-5 lists the attributes that are defined only for package subprograms.

**Table 8-5    Attributes Specific to Package Subprograms**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `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` |

## 8.2.3 Package Attributes

Lists and describes the attributes when a parameter is for a package.

Table 8-6 lists the attributes when a parameter is for a package (type `OCI_PTYPE_PKG`).

**Table 8-6 Attributes of Packages**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_LIST_PKG_TYPES` | Get a list of all types in an `OCI_PTYPE_PKG` package parameter handle. | `void *` |
| `OCI_ATTR_LIST_SUBPROGRAMS` | Subprogram list. See List Attributes. | `void *` |
| `OCI_ATTR_IS_INVOKER_RIGHTS` | Indicates that the package has invoker's rights? | `ub1` |

## 8.2.4 Type Attributes

Lists and describes the attributes when a parameter is for a type.

Table 8-7 lists the attributes when a parameter is for a type (type `OCI_PTYPE_TYPE`). These attributes are only valid if the application initialized the OCI process in `OCI_OBJECT` mode in a call to `OCIEnvCreate()`.

**Table 8-7 Attributes of Types**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_REF_TDO` | Returns the in-memory REF of the type descriptor object (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_TYPECODE` | Typecode. See Data Type Codes. Currently can be only `OCI_TYPECODE_OBJECT`, `OCI_TYPECODE_NAMEDCOLLECTION`, or `OCI_TYPECODE_RECORD`. | `OCITypeCode` |
| `OCI_ATTR_COLLECTION_TYPECODE` | Typecode of collection if type is collection; invalid otherwise. See Data Type Codes. Currently can be only `OCI_TYPECODE_VARRAY`, `OCI_TYPECODE_TABLE`, or `OCI_TYPECODE_ITABLE`. If this attribute is queried for a type that is not a collection, an error is returned. | `OCITypeCode` |
| `OCI_ATTR_IS_INCOMPLETE_TYPE` | Indicates that this is an incomplete type | `ub1` |
| `OCI_ATTR_IS_SYSTEM_TYPE` | Indicates that this is a system type | `ub1` |
| `OCI_ATTR_IS_PREDEFINED_TYPE` | Indicates that this is a predefined type | `ub1` |

**Table 8-7 (Cont.) Attributes of Types**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_IS_TRANSIENT_TYPE` | Indicates that this is a transient type | `ub1` |
| `OCI_ATTR_IS_SYSTEM_GENERATED_TYPE` | Indicates that 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. | `void *` |
| `OCI_ATTR_NUM_TYPE_ATTRS` | Number of type attributes | `ub2` |
| `OCI_ATTR_LIST_TYPE_ATTRS` | List of type attributes. See List Attributes. | `void *` |
| `OCI_ATTR_NUM_TYPE_METHODS` | Number of type methods | `ub2` |
| `OCI_ATTR_LIST_TYPE_METHODS` | List of type methods. See List Attributes. | `void *` |
| `OCI_ATTR_MAP_METHOD` | Map method of type. See Type Method Attributes. | `void *` |
| `OCI_ATTR_ORDER_METHOD` | Order method of type. See Type Method Attributes. | `void *` |
| `OCI_ATTR_IS_INVOKER_RIGHTS` | Indicates that the type has invoker's rights | `ub1` |
| `OCI_ATTR_NAME` | A pointer to a string that is the type attribute name | `OraText *` |
| `OCI_ATTR_PACKAGE_NAME` | A string with the package name if the attribute is a package type. | `OraText *` |
| `OCI_ATTR_SCHEMA_NAME` | A string with the schema name where the type has been created | `OraText *` |
| `OCI_ATTR_IS_FINAL_TYPE` | Indicates that this is a final type | `ub1` |
| `OCI_ATTR_IS_INSTANTIABLE_TYPE` | Indicates that this is an instantiable type | `ub1` |
| `OCI_ATTR_IS_SUBTYPE` | Indicates that this is a subtype | `ub1` |
| `OCI_ATTR_SUPERTYPE_SCHEMA_NAME` | Name of the schema that contains the supertype | `OraText *` |

**Table 8-7 (Cont.) Attributes of Types**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_SUPERTYPE_NAME` | Name of the supertype | `OraText *` |

**See Also:**

OCIEnvCreate()

## 8.2.5 Type Attribute Attributes

Lists and describes the attributes when a parameter is for an attribute of a type.

Table 8-8 lists the attributes when a parameter is for an attribute of a type (type `OCI_PTYPE_TYPE_ATTR`).

**Table 8-8 Attributes of Type Attributes**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `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 `NUMBER`s. | `ub2` |
| `OCI_ATTR_TYPECODE` | Typecode. See Data Type Codes. | `OCITypeCode` |
| `OCI_ATTR_DATA_TYPE` | The data type of the type attribute. See Data Type Codes. | `ub2` |
| `OCI_ATTR_NAME` | A pointer to a string that 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`; otherwise, it is a `NUMBER(precision, scale)`. When precision is 0, `NUMBER(precision, scale)` can be represented simply as `NUMBER`. | `ub1`<br>for explicit describe<br>`sb2`<br>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`; otherwise, it is a `NUMBER(precision, scale)`. When precision is 0, `NUMBER(precision, scale)` can be represented simply as `NUMBER`. | `sb1` |
| `OCI_ATTR_PACKAGE_NAME` | A string that is the package name of a type if it is a package type. | `OraText *` |

**Table 8-8 (Cont.) Attributes of Type Attributes**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_TYPE_NAME` | A string that is the type name. The returned value contains the type name if the data type is `SQLT_NTY` or `SQLT_REF`. If the data type is `SQLT_NTY`, the name of the named data type's type is returned. If the data type is `SQLT_REF`, the type name of the named data type pointed to by the `REF` is returned. | `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 or character type | `ub2` |
| `OCI_ATTR_CHARSET_FORM` | The character set form, if the type attribute is of a string or 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` |

## 8.2.6 Type Method Attributes

Lists and dsescribes the attributes when a parameter is for a method of a type.

Table 8-9 lists the attributes when a parameter is for a method of a type (type `OCI_PTYPE_TYPE_METHOD`).

**Table 8-9 Attributes of Type Methods**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `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 OCI_ATTR_LIST_ARGUMENTS Attribute, and List Attributes. | `void *` |
| `OCI_ATTR_IS_CONSTRUCTOR` | Indicates that method is a constructor | `ub1` |
| `OCI_ATTR_IS_DESTRUCTOR` | Indicates that method is a destructor | `ub1` |

**Table 8-9 (Cont.) Attributes of Type Methods**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_IS_OPERATOR` | Indicates that method is an operator | `ub1` |
| `OCI_ATTR_IS_SELFISH` | Indicates that method is selfish | `ub1` |
| `OCI_ATTR_IS_MAP` | Indicates that method is a map method | `ub1` |
| `OCI_ATTR_IS_ORDER` | Indicates that method is an order method | `ub1` |
| `OCI_ATTR_IS_RNDS` | Indicates that "Read No Data State" is set for method | `ub1` |
| `OCI_ATTR_IS_RNPS` | Indicates that "Read No Process State" is set for method | `ub1` |
| `OCI_ATTR_IS_WNDS` | Indicates that "Write No Data State" is set for method | `ub1` |
| `OCI_ATTR_IS_WNPS` | Indicates that "Write No Process State" is set for method | `ub1` |
| `OCI_ATTR_IS_FINAL_METHOD` | Indicates that this is a final method | `ub1` |
| `OCI_ATTR_IS_INSTANTIABLE_METHOD` | Indicates that this is an instantiable method | `ub1` |
| `OCI_ATTR_IS_OVERRIDING_METHOD` | Indicates that this is an overriding method | `ub1` |

## 8.2.7 Collection Attributes

Lists and describes the attributes when a parameter is for a collection type.

Table 8-10 lists the attributes when a parameter is for a collection type (type `OCI_PTYPE_COLL`).

**Table 8-10 Attributes of Collection Types**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `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 `NUMBER`s. | `ub2` |
| `OCI_ATTR_TYPECODE` | Typecode. See Data Type Codes. | `OCITypeCode` |
| `OCI_ATTR_DATA_TYPE` | The data type of the type attribute. See Data Type Codes. | `ub2` |
| `OCI_ATTR_NUM_ELEMS` | The number of elements in an array. It is only valid for collections that are arrays. | `ub4` |

**Table 8-10 (Cont.) Attributes of Collection Types**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_NAME` | A pointer to a string that 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`; otherwise, it is a `NUMBER(precision, scale)`. When precision is 0, `NUMBER(precision, scale)` can be represented simply as `NUMBER`. | `ub1`<br>for explicit describe<br>`sb2`<br>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`; otherwise, it is a `NUMBER(precision, scale)`. When precision is 0, `NUMBER(precision, scale)` can be represented simply as `NUMBER`. | `sb1` |
| `OCI_ATTR_PACKAGE_NAME` | A string that is the package name of a type if it is a package type. | `OraText *` |
| `OCI_ATTR_TYPE_NAME` | A string that is the type name. The returned value contains the type name if the data type is `SQLT_NTY` or `SQLT_REF`. If the data type is `SQLT_NTY`, the name of the named data type's type is returned. If the data type is `SQLT_REF`, the type name of the named data type pointed to by the REF is returned. | `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 type descriptor object (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 or character type | `ub2` |
| `OCI_ATTR_CHARSET_FORM` | The character set form, if the type attribute is of a string or character type | `ub1` |

## 8.2.8 Synonym Attributes

Lists and describes the attributes when a parameter is for a synonym.

Table 8-11 lists the attributes when a parameter is for a synonym (type `OCI_PTYPE_SYN`).

**Table 8-11 Attributes of Synonyms**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_OBJID` | Object ID | `ub4` |

**Table 8-11 (Cont.) Attributes of Synonyms**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `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 *` |

## 8.2.9 Sequence Attributes

Lists and describes the attributes when a parameter is for a sequence.

Table 8-12 lists the attributes when a parameter is for a sequence (type `OCI_PTYPE_SEQ`).

**Table 8-12 Attributes of Sequences**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `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

## 8.2.10 Column Attributes

Lists and describes the attributes when a parameter is for a column of a table or view.

> **Note:**
>
> For `BINARY_FLOAT` and `BINARY_DOUBLE`:
>
> If `OCIDescribeAny()` is used to retrieve the column data type (`OCI_ATTR_DATA_TYPE`) for `BINARY_FLOAT` or `BINARY_DOUBLE` columns in a table, it returns the internal data type code.
>
> The SQLT codes corresponding to the internal data type codes for `BINARY_FLOAT` and `BINARY_DOUBLE` are `SQLT_IBFLOAT` and `SQLT_IBDOUBLE`.

The following table lists the attributes when a parameter is for a column of a table or view (type `OCI_PTYPE_COL`).

**Table 8-13 Attributes of Columns of Tables or Views**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_CHAR_USED` | Returns the type of length semantics of the column. Zero (0) means byte-length semantics and 1 means character-length semantics. See Character-Length Semantics Support in Describe Operations. | `ub1` |
| `OCI_ATTR_CHAR_SIZE` | Returns the column character length that 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 Describe Operations. | `ub2` |

**Table 8-13 (Cont.) Attributes of Columns of Tables or Views**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_COLLATION_ID` | Returns the derived collation ID of the column. The `OCI_ATTR_COLLATION_ID` attribute is valid only for select-list describes based on a statement handle for a query and not for `OCIDescribeAny()` describes. It is relevant only for select-list items of character data types. For other data types the value is always `OCI_COLLATION_NONE`. Table 8-14 describes the predefined constraints for a number of commonly used collation IDs.<br>**Note:**<br>You can use the SQL built-in functions `NLS_COLLATION_ID` and `NLS_COLLATION_NAME` to map between collation IDs returned for this attribute and collation names used in SQL syntax.<br>**See Also:**<br>• OCI_ATTR_COLLATION_ID<br>• *Oracle Database Globalization Support Guide* for information on how this set collation influences query processing | `ub4 *` |

**Table 8-13 (Cont.) Attributes of Columns of Tables or Views**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_COL_PROPERTIES` | Returns describe data regarding certain column properties. The following are the flags available in `OCI_ATTR_COL_PROPERTIES`:<br>• `OCI_ATTR_COL_PROPERTY_IS_IDENTITY`<br>• `OCI_ATTR_COL_PROPERTY_IS_GEN_ALWAYS`<br>  – 1 - represents it is `ALWAYS GENERATED`<br>  – 0 - represents `GENERATED BY DEFAULT`<br>• `OCI_ATTR_COL_PROPERTY_IS_GEN_BY_DEF_ON_NULL`<br>• `OCI_ATTR_COL_PROPERTY_IS_LPART`<br>• `OCI_ATTR_COL_PROPERTY_IS_CONID`<br>The following is a sample usage: | `ub8` |

```
OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
  (dvoid*) &col_prop, (ub4 *) 0,(ub4) OCI_ATTR_COL_PROPERTIES,
  (OCIError *) errhp  ));
if(col_prop & OCI_ATTR_COL_PROPERTY_IS_IDENTITY)
  printf("Identity Column \n");
if(col_prop & OCI_ATTR_COL_PROPERTY_IS_GEN_ALWAYS)
  printf("Column is always generated\n");
if(col_prop & OCI_ATTR_COL_PROPERTY_IS_GEN_BY_DEF_ON_NULL)
  printf("Column is generated by default on NULL\n");
if(col_prop & OCI_ATTR_COL_PROPERTY_IS_LPART)
  printf("Column is an implicitly generated logical
partitioning column for container_map enabled objects\n");
if(col_prop & OCI_ATTR_COL_PROPERTY_IS_CONID)
  printf("Column is a CON_ID column implicitly generated by
CONTAINERS() or is an ORIGIN_CON_ID column implicitly
generated for Extended Data Link\n");
```

See *Oracle Database SQL Language Reference* for more information about the SQL syntax to specify these properties in an identity clause for a column definition in the `CREATE TABLE` statement.

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_INVISIBLE_COL` | Returns whether a column is invisible or not. A value of `TRUE` indicates the column is an invisible column. Otherwise, the value `FALSE` is returned. See "Describing Each Column to Know Whether It Is an Invisible Column" for an example. | `ub1` |
| `OCI_ATTR_DATA_SIZE` | The maximum size of the column. This length is returned in bytes and not characters for strings and raws. It returns 22 for `NUMBER`s. | `ub2` |
| `OCI_ATTR_DATA_TYPE` | The data type of the column. See Data Type Codes. | `ub2` |
| `OCI_ATTR_NAME` | A pointer to a string that is the column name | `OraText *` |

**Table 8-13 (Cont.) Attributes of Columns of Tables or Views**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_PRECISION` | The precision of numeric columns. If the precision is nonzero and scale is -127, then it is a `FLOAT`; otherwise, it is a `NUMBER(precision, scale)`. When precision is 0, `NUMBER(precision, scale)` can be represented simply as `NUMBER`. | `ub1`<br>for explicit describe<br>`sb2`<br>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`; otherwise, it is a `NUMBER(precision, scale)`. 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. Does not return a correct value for a `CUBE` or `ROLLUP` operation. | `ub1` |
| `OCI_ATTR_TYPE_NAME` | Returns a string that is the type name. The returned value contains the type name if the data type is `SQLT_NTY` or `SQLT_REF`. If the data type is `SQLT_NTY`, the name of the named data type's type is returned. If the data type is `SQLT_REF`, the type name of the named data type pointed to by the `REF` is returned. | `OraText *` |
| `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 or character type | `ub2` |
| `OCI_ATTR_CHARSET_FORM` | The character set form, if the column is of a string or character type | `ub1` |

Table 8-14 describes predefined constants for a number of commonly used collation IDs, which can be returned for the attribute `OCI_ATTR_COLLATION_ID`. The constants are listed with their values (in parenthesis) and the SQL names of the corresponding collations.

**Table 8-14 Predefined Collation IDs, Their `ub4` Values (in parenthesis), and Their SQL Names**

| Collation ID (Value) | SQL Name |
|---|---|
| OCI_COLLATION_NONE (0) | Undefined collation; no collation has been specified. |
| OCI_COLLATION_NLS_COMP (16382) | USING_NLS_COMP |
| OCI_COLLATION_NLS_SORT (16381) | USING_NLS_SORT |
| OCI_COLLATION_NLS_SORT_CI (16380) | USING_NLS_SORT_CI |
| OCI_COLLATION_NLS_SORT_AI (16379) | USING_NLS_SORT_AI |

**Table 8-14 (Cont.) Predefined Collation IDs, Their ub4 Values (in parenthesis), and Their SQL Names**

| Collation ID (Value) | SQL Name |
|---|---|
| OCI_COLLATION_NLS_SORT_CS (16378) | USING_NLS_SORT_CS |
| OCI_COLLATION_BINARY (16383) | BINARY |
| OCI_COLLATION_BINARY_CI (147455) | BINARY_CI |
| OCI_COLLATION_BINARY_AI (81919) | BINARY_AI |

**See Also:**

OCIDescribeAny()

## 8.2.11 Argument and Result Attributes

Lists and describes the attributes when a parameter is for an argument of a procedure or function.

Table 8-15 lists the 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`).

**Table 8-15 Attributes of Arguments and Results**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_NAME` | Returns a pointer to a string that 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 Data Type Codes. | `OCITypeCode` |
| `OCI_ATTR_DATA_TYPE` | The data type of the argument. See Data Type Codes. | `ub2` |
| `OCI_ATTR_DATA_SIZE` | The size of the data type of the argument. This length is returned in bytes and not characters for strings and raws. It returns 22 for `NUMBER`s. | `ub2` |
| `OCI_ATTR_PRECISION` | The precision of numeric arguments. If the precision is nonzero and scale is -127, then it is a `FLOAT`; otherwise, it is a `NUMBER(precision, scale)`. When precision is 0, `NUMBER(precision, scale)` can be represented simply as `NUMBER`. | `sb1` for explicit describe<br>`sb2` for implicit describe |

**Table 8-15 (Cont.) Attributes of Arguments and Results**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_SCALE` | The scale of numeric arguments. If the precision is nonzero and scale is -127, then it is a `FLOAT`; otherwise, it is a `NUMBER(precision, scale)`. When precision is 0, `NUMBER(precision, scale)` can be represented simply as `NUMBER`. | `sb1` |
| `OCI_ATTR_LEVEL` | The data type levels. This attribute always returns zero. | `ub2` |
| `OCI_ATTR_HAS_DEFAULT` | Indicates whether an argument has a default | `ub1` |
| `OCI_ATTR_LIST_ARGUMENTS` | The list of arguments at the next level (when the argument is of a record or table type) | `void *` |
| `OCI_ATTR_IOMODE` | Indicates the argument mode:<br>0 is IN (`OCI_TYPEPARAM_IN`),<br>1 is OUT (`OCI_TYPEPARAM_OUT`),<br>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 that is the type name or the package name for package local types. The returned value contains the type name if the data type is `SQLT_NTY` or `SQLT_REF`. If the data type is `SQLT_NTY`, the name of the named data type's type is returned. If the data type is `SQLT_REF`, the type name of the named data type pointed to by the REF is returned. | `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 for package local types | `OraText *` |
| `OCI_ATTR_SUB_NAME` | For `SQLT_NTY` or `SQLT_REF`, returns a string with the type name, for 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 for package local types, when the package is remote. | `OraText *` |
| `OCI_ATTR_REF_TDO` | Returns the REF of the type descriptor object (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 or character type | `ub2` |
| `OCI_ATTR_CHARSET_FORM` | Returns the character set form if the argument is of a string or character type | `ub1` |

## 8.2.12 List Attributes

Lists and describes the attributes when the parameter is for a list of columns, arguments, and subprograms, or fields of a package record type.

When a parameter is for a list of columns, arguments, and subprograms, or fields of a package record type (type `OCI_PTYPE_LIST`), it has the type-specific attributes and handles (parameters) shown in Table 8-16.

The list has an `OCI_ATTR_LTYPE` attribute that designates the list type. Table 8-16 lists the possible values and their lower bounds when traversing the list.

**Table 8-16 List Attributes**

| List Attribute | Description | Lower Bound |
|---|---|---|
| `OCI_LTYPE_COLUMN` | Column list | 1 |
| `OCI_LTYPE_ARG_PROC` | Procedure argument list | 1 |
| `OCI_LTYPE_ARG_FUNC` | Function argument list | 0 |
| `OCI_LTYPE_SUBPRG` | Subprogram list | 0 |
| `OCI_LTYPE_TYPE_ATTR` | Type attribute list | 1 |
| `OCI_LTYPE_TYPE_METHOD` | Type method list | 1 |
| `OCI_LTYPE_TYPE_ARG_PROC` | Type method without result argument list | 0 |
| `OCI_LTYPE_TYPE_ARG_FUNC` | Type method without result argument list | 1 |
| `OCI_LTYPE_SCH_OBJ` | Object list within a schema | 0 |
| `OCI_LTYPE_DB_SCH` | Schema list within a database | 0 |

The list has an `OCI_ATTR_NUM_PARAMS` attribute, which tells the number of elements in the list.

Each list has `LowerBound` ... `OCI_ATTR_NUM_PARAMS` parameters. `LowerBound` is the value in the Lower Bound column of Table 8-16. For a function argument list, position 0 has a parameter for the return value (type `OCI_PTYPE_ARG`).

## 8.2.13 Schema Attributes

Lists and describes the attributes when a parameter is for a schema type.

Table 8-17 lists the attributes when a parameter is for a schema type (type `OCI_PTYPE_SCHEMA`).

**Table 8-17 Attributes Specific to Schemas**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_LIST_OBJECTS` | List of objects in the schema | `OCIParam *` |

## 8.2.14 Database Attributes

Lists and describes the attributes when a parameter is for a database type.

Table 8-18 lists the attributes when a parameter is for a database type (type `OCI_PTYPE_DATABASE`).

**Table 8-18 Attributes Specific to Databases**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_VERSION` | Database version | `OraText *` |
| `OCI_ATTR_CHARSET_ID` | Database character set ID from the server handle | `ub2` |
| `OCI_ATTR_NCHARSET_ID` | Database national 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` |
| `OCI_ATTR_CURSOR_COMMIT_BEHAVIOR` | How a `COMMIT` operation affects cursors and prepared statements in the database. Values are:<br>`OCI_CURSOR_OPEN` - Preserve cursor state as before the commit operation.<br>`OCI_CURSOR_CLOSED` - Cursors are closed on `COMMIT`, but the application can still reexecute the statement without preparing it again. | `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` |

## 8.2.15 Rule Attributes

Lists and describes the attributes when a parameter is for a rule.

Table 8-19 lists the attributes when a parameter is for a rule (type `OCI_PTYPE_RULE`).

**Table 8-19 Attributes Specific to Rules**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_CONDITION` | Rule condition | `OraText *` |
| `OCI_ATTR_EVAL_CONTEXT_OWNER` | Owner name of the evaluation context associated with the rule, if any | `OraText *` |
| `OCI_ATTR_EVAL_CONTEXT_NAME` | Object name of the 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`) | `void *` |

## 8.2.16 Rule Set Attributes

Lists and describes the attributes when a parameter is for a rule set.

Table 8-20 lists the attributes when a parameter is for a rule set (type `OCI_PTYPE_RULE_SET`).

**Table 8-20 Attributes Specific to Rule Sets**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_EVAL_CONTEXT_OWNER` | Owner name of the evaluation context associated with the rule set, if any | `OraText *` |
| `OCI_ATTR_EVAL_CONTEXT_NAME` | Object name of the 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`) | `void *` |

## 8.2.17 Evaluation Context Attributes

Lists and describes the attributes when a parameter is for an evaluation context.

Table 8-21 lists the attributes when a parameter is for an evaluation context (type `OCI_PTYPE_EVALUATION_CONTEXT`).

**Table 8-21 Attributes Specific to Evaluation Contexts**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `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`) | `void *` |
| `OCI_ATTR_LIST_VARIABLE_TYPES` | List of variable types in the evaluation context (type `OCI_PTYPE_LIST`) | `void *` |

## 8.2.18 Table Alias Attributes

Lists and describes the attributes when a parameter is for a table alias.

Table 8-22 lists the attributes when a parameter is for a table alias (type `OCI_PTYPE_TABLE_ALIAS`).

**Table 8-22 Attributes Specific to Table Aliases**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_NAME` | Table alias name | `OraText *` |
| `OCI_ATTR_TABLE_NAME` | Table name associated with the alias | `OraText *` |

## 8.2.19 Variable Type Attributes

Lists and describes the attributes when a parameter is for a variable.

Table 8-23 lists the attributes when a parameter is for a variable (type `OCI_PTYPE_VARIABLE_TYPE`).

**Table 8-23 Attributes Specific to Variable Types**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `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 *` |
| `OCI_ATTR_VAR_METHOD_FUNCTION` | Variable method function associated with the variable, if any | `OraText *` |

### 8.2.20 Name Value Attributes

Lists and describes the attributes when a parameter is for a name-value pair.

Table 8-24 lists the attributes when a parameter is for a name-value pair (type `OCI_PTYPE_NAME_VALUE`).

**Table 8-24 Attributes Specific to Name-Value Pair**

| Attribute | Description | Attribute Data Type |
|---|---|---|
| `OCI_ATTR_NAME` | Name | `OraText *` |
| `OCI_ATTR_VALUE` | Value | `OCIAnyData*` |

## 8.3 Character-Length Semantics Support in Describe Operations

Query and column information are supported with character-length semantics.

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`, which 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. Zero (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()`.

This section includes the following topics:

- Implicit Describing
- Explicit Describing

**See Also:**

- OCIStmtExecute()
- OCIDescribeAny()

## 8.3.1 Implicit Describing

If the database column was created using character-length semantics, then the implicit describe information contains 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 is always large enough to hold all the data, as many as `OCI_ATTR_CHAR_SIZE` number of characters. The `OCI_ATTR_DATA_SIZE` is usually set to (`OCI_ATTR_CHAR_SIZE`)*(the client's maximum number of bytes) for each character value.

If the database column was created with byte-length semantics, then for the implicit describe (it behaves exactly as it does before release 9.0) the `OCI_ATTR_DATA_SIZE` value returned is (column's byte length)*(the maximum conversion ratio between the client and server's character set). That is, the column byte length divided by the server's maximum number of bytes for each character multiplied by the client's maximum number of bytes for each character. The `OCI_ATTR_CHAR_USED` value is 0 and the `OCI_ATTR_CHAR_SIZE` value is set to the same value as `OCI_ATTR_DATA_SIZE`.

## 8.3.2 Explicit Describing

Explicit describes of tables have three attributes: `OCI_ATTR_DATA_SIZE`, `OCI_ATTR_CHAR_SIZE`, and `OCI_ATTR_CHAR_USED`.

Explicit describes of tables have the following attributes:

- `OCI_ATTR_DATA_SIZE` gets the column's size in bytes, as it appears in the server
- `OCI_ATTR_CHAR_SIZE` indicates the length of the column in characters
- `OCI_ATTR_CHAR_USED`, is a flag that indicates how the column was created, as described previously in terms of the type of length semantics of the column

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 prevents you from violating the size constraint for the column.

This section includes the following topic: Client and Server Compatibility Issues for Describing.

**See Also:**

IN Binds

### 8.3.2.1 Client and Server Compatibility Issues for Describing

Character-length semantics depends on the release of the server or client. It is best described when both server and client are Oracle9*i* or later. Otherwise, compatibility issues as described result.

When an Oracle9*i* or later client talks to an Oracle8*i* or earlier server, it behaves 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.

# 8.4 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 Database installation.

- Retrieving Column Data Types for a Table
- Describing the Stored Procedure
- Retrieving Attributes of an Object Type
- Retrieving the Collection Element's Data Type of a Named Collection Type
- Describing with Character-Length Semantics
- Describing Each Column to Know Whether It Is an Invisible Column

> **See Also:**
>
> - OCI Demonstration Programs for additional information about the demonstration programs
> - OCIDescribeAny()

## 8.4.1 Retrieving Column Data Types for a Table

Illustrates the use of an explicit describe that retrieves the column data types for a table.

Example 8-2 illustrates the use of an explicit describe that retrieves the column data types for a table.

**Example 8-2    Using an Explicit Describe to Retrieve Column Data Types 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((void *)envhp, (void **)&dschp,
        (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (void **)0);

/* get the describe handle for the table */
if (OCIDescribeAny(svch, errh, (void *)objptr, objp_len, OCI_OTYPE_NAME, 0,
     OCI_PTYPE_TABLE, dschp))
   return OCI_ERROR;

/* get the parameter handle */
if (OCIAttrGet((void *)dschp, OCI_HTYPE_DESCRIBE, (void *)&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((void *)parmh, OCI_DTYPE_PARAM, (void *)&numcols, (ub4 *)0,
     OCI_ATTR_NUM_COLS, errh))
    return OCI_ERROR;

/* get the handle to the column list of the table */
if (OCIAttrGet((void *)parmh, OCI_DTYPE_PARAM, (void *)&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((void *)collsthd, OCI_DTYPE_PARAM, errh, (void **)&colhd,
(ub4)i))
        return OCI_ERROR;

    /* for example, get data type for ith column */
    coltyp = 0;
    if (OCIAttrGet((void *)colhd, OCI_DTYPE_PARAM, (void *)&coltyp, (ub4 *)0,
       OCI_ATTR_DATA_TYPE, errh))
        return OCI_ERROR;

    /* Retrieve the length semantics for the column */
    char_semantics = 0;
    OCIAttrGet((void*) colhd, (ub4) OCI_DTYPE_PARAM,
                 (void*) &char_semantics,(ub4 *) 0, (ub4) OCI_ATTR_CHAR_USED,
                 (OCIError *) errh);

    col_width = 0;
    if (char_semantics)
        /* Retrieve the column width in characters */
        OCIAttrGet((void*) colhd, (ub4) OCI_DTYPE_PARAM,
                 (void*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_SIZE,
                 (OCIError *) errh);
    else
        /* Retrieve the column width in bytes */
        OCIAttrGet((void*) colhd, (ub4) OCI_DTYPE_PARAM,
                 (void*) &col_width,(ub4 *) 0, (ub4) OCI_ATTR_DATA_SIZE,
                 (OCIError *) errh);
}

if (dschp)
    OCIHandleFree((void *) dschp, OCI_HTYPE_DESCRIBE);
...
```

## 8.4.2 Describing the Stored Procedure

The steps required to describe type methods (also divided into functions and procedures) are identical to those of regular PL/SQL functions and procedures.

The difference between a procedure and a function is that the latter has a return type at position 0 in the argument list, whereas the former has no argument associated with position 0 in the argument list. Note that procedures and functions can take the default types of objects as arguments. Consider the following procedure:

```
P1 (arg1 emp.sal%type, arg2 emp%rowtype)
```

In Example 8-3, 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 1 and 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.

**Example 8-3    Describing the Stored Procedure**

```
...
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((void *)envhp, (void **)&dschp,
        (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (void **)0);

/* get the describe handle for the procedure */
if (OCIDescribeAny(svch, errh, (void *)objptr, objp_len, OCI_OTYPE_NAME, 0,
     OCI_PTYPE_PROC, dschp))
   return OCI_ERROR;

/* get the parameter handle */
if (OCIAttrGet((void *)dschp, OCI_HTYPE_DESCRIBE, (void *)&parmh, (ub4 *)0,
         OCI_ATTR_PARAM, errh))
```

```
    return OCI_ERROR;

/* Get the number of arguments and the arg list */
if (OCIAttrGet((void *)parmh, OCI_DTYPE_PARAM, (void *)&arglst,
       (ub4 *)0, OCI_ATTR_LIST_ARGUMENTS, errh))
    return OCI_ERROR;

if (OCIAttrGet((void *)arglst, OCI_DTYPE_PARAM, (void *)&numargs, (ub4 *)0,
    OCI_ATTR_NUM_PARAMS, errh))
    return OCI_ERROR;

/* For a procedure, you begin with i = 1; for a
function, you begin with i = 0. */

for (i = 1; i <= numargs; i++) {
  OCIParamGet ((void *)arglst, OCI_DTYPE_PARAM, errh, (void **)&arg, (ub4)i);
  namelen = 0;
  OCIAttrGet((void *)arg, OCI_DTYPE_PARAM, (void *)&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((void *)arg, OCI_DTYPE_PARAM, (void *)&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((void *)arglst1, OCI_DTYPE_PARAM, (void *)&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((void *)arglst1, OCI_DTYPE_PARAM, errh, (void **)&arg1,
          (ub4)j);
      namelen1 = 0;
      OCIAttrGet((void *)arg1, OCI_DTYPE_PARAM, (void *)&name1, (ub4 *)&namelen1,
        OCI_ATTR_NAME, errh);
    }
  }
}

if (dschp)
    OCIHandleFree((void *) dschp, OCI_HTYPE_DESCRIBE);
...
```

## 8.4.3 Retrieving Attributes of an Object Type

Illustrates the use of an explicit describe on a named object type.

Example 8-4 illustrates the use of an explicit describe on a named object type. It illustrates how you can describe an object by its name or by its object reference (`OCIRef`). The following code fragment attempts to retrieve the data type value of each of the object type's attributes.

### Example 8-4 Using an Explicit Describe on a Named Object Type

```
...
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((void *)envh, (void **)&dschp,
                  (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (void **)0))
   return OCI_ERROR;

/* get the describe handle for the type */
if (describe_by_name) {
  if (OCIDescribeAny(svch, errh, (void *)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, (void*)type_ref, 0, OCI_OTYPE_REF,
        0, OCI_PTYPE_TYPE, dschp))
       return OCI_ERROR;
}

/* get the parameter handle */
if (OCIAttrGet((void *)dschp, OCI_HTYPE_DESCRIBE, (void *)&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 OCIAttrGet */

/* get the number of attributes in the type */
if (OCIAttrGet((void *)parmh, OCI_DTYPE_PARAM, (void *)&numattrs, (ub4 *)0,
    OCI_ATTR_NUM_TYPE_ATTRS, errh))
    return OCI_ERROR;

/* get the handle to the attribute list of the type */
if (OCIAttrGet((void *)parmh, OCI_DTYPE_PARAM, (void *)&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((void *)attrlsthd, OCI_DTYPE_PARAM, errh, (void **)&attrhd, i))
      return OCI_ERROR;
/* for example, get data type and typecode for attribute; note that
OCI_ATTR_DATA_TYPE returns the SQLT code, whereas OCI_ATTR_TYPECODE returns the
Oracle Type System typecode. */

datatype = 0;
if (OCIAttrGet((void *)attrhd, OCI_DTYPE_PARAM, (void *)&datatype, (ub4 *)0,
               OCI_ATTR_DATA_TYPE,errh))
    return OCI_ERROR;

typecode = 0;
if (OCIAttrGet((void *)attrhd, OCI_DTYPE_PARAM,(void *)&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((void *)envh,(void**)&nested_dschp,
    (ub4)OCI_HTYPE_DESCRIBE,(size_t)0, (void **)0))
    return OCI_ERROR;

  if (OCIAttrGet((void *)attrhd, OCI_DTYPE_PARAM,
        (void *)&attr_type_ref, (ub4 *)0, OCI_ATTR_REF_TDO,errh))
    return OCI_ERROR;

   OCIDescribeAny(svch, errh,(void*)attr_type_ref, 0,
        OCI_OTYPE_REF, 0, OCI_PTYPE_TYPE, nested_dschp);
    /* go on describing the attribute type... */
}
}

if (dschp)
    OCIHandleFree((void *) dschp, OCI_HTYPE_DESCRIBE);
...
```

## 8.4.4 Retrieving the Collection Element's Data Type of a Named Collection Type

Illustrates the use of an explicit describe on a named collection type.

Example 8-5 illustrates the use of an explicit describe on a named collection type.

**Example 8-5 Using 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;
void *collection_element_parmh = (void *) 0;
OCIDescribe *dschp = (OCIDescribe *) 0;      /* describe handle */
OCIParam *parmh = (OCIParam *) 0;         /* parameter handle */

/* allocate describe handle */
```

```
if (OCIHandleAlloc((void *)envh, (void **)&dschp,
                  (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (void **)0))
   return OCI_ERROR;

/* get the describe handle for the type */
if (describe_by_name) {
  if (OCIDescribeAny(svch, errh, (void *)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, (void*)type_ref, 0, OCI_OTYPE_REF,
        0, OCI_PTYPE_TYPE, dschp))
       return OCI_ERROR;
}

/* get the parameter handle */
if (OCIAttrGet((void *)dschp, OCI_HTYPE_DESCRIBE, (void *)&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((void *)parmh, OCI_DTYPE_PARAM,(void *)&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: OCI_TYPECODE_VARRAY or OCI_TYPECODE_TABLE */
  collection_typecode = 0;
  if (OCIAttrGet((void *)parmh, OCI_DTYPE_PARAM, (void *)&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((void *)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((void *)collection_element_parmh, OCI_DTYPE_PARAM,
          (void *)&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((void *)collection_element_parmh, OCI_DTYPE_PARAM,
        (void *)&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((void *) dschp, OCI_HTYPE_DESCRIBE);
...
```

## 8.4.5 Describing with Character-Length Semantics

Shows a loop that retrieves the column names and data types corresponding to a query following query execution.

Example 8-6 shows a loop that retrieves the column names and data types corresponding to a query following query execution. The query was associated with the statement handle by a prior call to OCIStmtPrepare2().

**Example 8-6    Using a Parameter Descriptor to Retrieve the Data Types, Column Names, and Character-Length Semantics**

```
...
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, OCIStmtPrepare2(svchp, &stmthp, errhp, (OraText *)sqlstmt,
                    (ub4)strlen((char *)sqlstmt), NULL, 0,
                    (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((void *)stmthp, OCI_HTYPE_STMT, errhp,
               (void **)&mypard, (ub4) counter);
/* Loop only if a descriptor was successfully retrieved for
   current position, starting at 1 */
while (parm_status == OCI_SUCCESS) {
   /* Retrieve the data type attribute */
   checkerr(errhp, OCIAttrGet((void*) mypard, (ub4) OCI_DTYPE_PARAM,
           (void*) &dtype,(ub4 *) 0, (ub4) OCI_ATTR_DATA_TYPE,
           (OCIError *) errhp  ));
   /* Retrieve the column name attribute */
   col_name_len = 0;
   checkerr(errhp, OCIAttrGet((void*) mypard, (ub4) OCI_DTYPE_PARAM,
           (void**) &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((void*) mypard, (ub4) OCI_DTYPE_PARAM,
           (void*) &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((void*) mypard, (ub4) OCI_DTYPE_PARAM,
                (void*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_SIZE,
                (OCIError *) errhp  ));
    else
        /* Retrieve the column width in bytes */
        checkerr(errhp, OCIAttrGet((void*) mypard, (ub4) OCI_DTYPE_PARAM,
                (void*) &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((void *)stmthp, OCI_HTYPE_STMT, errhp,
           (void **)&mypard, (ub4) counter);
} /* while */
...
```

## 8.4.6 Describing Each Column to Know Whether It Is an Invisible Column

Illustrates the use of invisible column properties and checking each column to determine if it is an invisible column.

The following code example illustrates the use of invisible column properties and checking each column to determine if it is an invisible column. See the `OCI_ATTR_INVISIBLE_COL` attribute description in the table in Column Attributes for more information.

**Example 8-7    Checking for Invisible Columns**

```
.....
.....
  checkerr(errhp, OCIHandleAlloc((dvoid *) envhp, (dvoid **) &dschp,
                         (ub4) OCI_HTYPE_DESCRIBE,
                         (size_t) 0, (dvoid **) 0));
  /* Set the invisible column attribute to get the invisible column(s). */
  checkerr(errhp, OCIAttrSet(dschp, OCI_HTYPE_DESCRIBE, &invscols, 0,
                           OCI_ATTR_SHOW_INVISIBLE_COLUMNS, errhp));

  if ((retval = OCIDescribeAny(svchp, errhp, (dvoid *)tablename,
                               (ub4) strlen((char *) tablename),
                               OCI_OTYPE_NAME, (ub1)1,
                               OCI_PTYPE_TABLE, dschp)) != OCI_SUCCESS)
  {
    if (retval == OCI_NO_DATA)
    {
      printf("NO DATA: OCIDescribeAny on %s\n", tablename);
    }
    else                                                 /* OCI_ERROR */
    {
      printf( "ERROR: OCIDescribeAny on %s\n", tablename);
      checkerr(errhp, retval);
      return;
    }
  }
  else
  {
    ub1 colIsInv;
    /* Get the parameter descriptor. */
```

```
 checkerr (errhp, OCIAttrGet((dvoid *)dschp, (ub4)OCI_HTYPE_DESCRIBE,
                      (dvoid *)&parmp, (ub4 *)0, (ub4)OCI_ATTR_PARAM,
                      (OCIError *)errhp));

 /* Get the attributes of the table. */
 checkerr (errhp, OCIAttrGet((dvoid*) parmp, (ub4) OCI_DTYPE_PARAM,
                      (dvoid*) &objid, (ub4 *) 0,
                      (ub4) OCI_ATTR_OBJID, (OCIError *)errhp));
 /* Get the column list of the table. */
 checkerr (errhp, OCIAttrGet((dvoid*) parmp, (ub4) OCI_DTYPE_PARAM,
                      (dvoid*) &collst, (ub4 *) 0,
                      (ub4) OCI_ATTR_LIST_COLUMNS, (OCIError *)errhp));
 /* Get the number of columns. */
 checkerr (errhp, OCIAttrGet((dvoid*) parmp, (ub4) OCI_DTYPE_PARAM,
                      (dvoid*) &numcols, (ub4 *) 0,
                      (ub4) OCI_ATTR_NUM_COLS, (OCIError *)errhp));

/* Now describe each column to know whether it is a invisible column or not. */

  for (pos = 1; pos <= parmcnt; pos++)
    {
    /* Get the parameter descriptor for each column. */
    checkerr (errhp, OCIParamGet((dvoid *)parmp, (ub4)OCI_DTYPE_PARAM, errhp,
                    (dvoid *)&parmdp, (ub4) pos));
    .....
    .....

    checkerr (errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
                      (dvoid*) &colIsInv, (ub4 *) 0,
                      (ub4) OCI_ATTR_INVISIBLE_COL, (OCIError *)errhp));

    .....
    .....

    }

}
 .....
 .....
```

# 9

# LOB and BFILE Operations

This chapter describes LOB and BFILE operations.

This chapter contains these topics:

- About Using OCI Functions for LOBs
- About Creating and Modifying Persistent LOBs
- About Associating a BFILE in a Table with an Operating System File
- LOB Attributes of an Object
- Array Interface for LOBs
- About Using LOBs of Size Greater than 4 GB
- LOB and BFILE Functions in OCI
- Temporary LOB Support
- Prefetching of LOB Data, Length, and Chunk Size
- Options of SecureFiles LOBs

## 9.1 About Using OCI Functions for LOBs

OCI includes a set of functions for performing operations on large objects (LOBs) in a database.

Persistent LOBs (`BLOB`s, `CLOB`s, `NCLOB`s) 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 Oracle database. `BFILE`s are large data objects stored in the server's operating system files outside the database tablespaces.

OCI also provides support for temporary LOBs, which can be used like local variables for operating on LOB data.

`BFILE`s are read-only. Oracle Database supports only binary `BFILE`s.

Beginning with Oracle Database 12c Release 2 (12.2), most LOB functions support Application Continuity.

Beginning with Oracle Database 12c Release 2 (12.2), OCI LOB APIs support distributed operations on remote LOBs (CLOB, BLOB) except for OCI LOB APIs for BFILES. This includes support for certain queries that select a remote LOB locator that in previous releases returned an error. The only restriction is that all LOB APIs that take in two locators should have both LOBs collocated at one database. The following LOB functions throw an error when a remote locator is passed to it: `OCILobAssign()`, `OCILobLocatorAssign()`, `OCILobEnableBuffering()`, `OCILobDisableBuffering()`, `OCILobFlushBuffer()`, `OCILobArrayRead()`, `OCILobArrayWrite()`, and `OCILobLoadFromFile2()`.

**See Also:**

- 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 SecureFiles and Large Objects Developer's Guide*
- When Application Continuity in OCI Can Fail Over for a complete list.
- Deprecated Lob Functions for information about support for Application Continuity also includes some deprecated LOB functions for compatibility.

## 9.2 About 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:

- 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, to bind 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 because neither the remote server nor the local server is of a release earlier than Oracle Database 10*g* Release 2. The data interface only supports data size up to 2 GB – 1, the maximum size of an `sb4` data type.

- 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 perform a `SELECT`...`FOR UPDATE` operation on 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.

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` statement, for example), because the commit closes the transaction.

> **See Also:**
>
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* chapter about data interface for persistent LOBs for more information and examples
> - About Binding LOB Data for usage and examples for both `INSERT` and `UPDATE`
> - OCIDescriptorAlloc()
> - OCIAttrSet()

## 9.3 About 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` statement without bind variables, and `OCILobFileSetName()` is used for bind variables.

> **See Also:**
>
> - OCILobFileSetName()
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about the `BFILENAME` function

## 9.4 LOB Attributes of an Object

An OCI application can use the `OCIObjectNew()` function to create a persistent or transient object with a LOB attribute.

This section includes the following topics:

- Writing to a LOB Attribute of an Object
- Transient Objects with LOB Attributes

**See Also:**

OCIObjectNew()

## 9.4.1 Writing to a LOB Attribute of an Object

It is possible to use 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" (modified).
3. Flush the object, thereby inserting a row into the table.
4. Repin the latest version of the object (or refresh the object), thereby retrieving the object from the database and acquiring a valid locator for the LOB.
5. Call `OCIObjectWrite2()` using the LOB locator in the object to write the data.

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:**

- OCI Object-Relational Programming and the chapters that follow it for more information about objects
- About Binding LOB Data for usage and examples for both `INSERT` and `UPDATE statements`
- About Defining LOB Data for usage and examples of `SELECT` statements
- OCIObjectNew()
- OCILobWrite2()

## 9.4.2 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 does 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`.

**See Also:**

OCIObjectNew()

## 9.5 Array Interface for LOBs

You can use the OCI array interface with LOBs, just as with any other data type.

There are two ways of using the array interface.

- 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.

- Using the LOB locator

  When using the LOB locator you must allocate the descriptors, as shown in the following code example.

  Using the LOB Locator and Allocating 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], ...);
```

**See Also:**

- About Binding LOB Data for usage and examples for both `INSERT` and `UPDATE statements`
- About Defining LOB Data for usage and examples of `SELECT` statements

## 9.6 About Using LOBs of Size Greater than 4 GB

Starting with Oracle Database 10*g* 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 enables you to create tablespaces with block sizes different from the database block size. The maximum size of a LOB depends on the size of the tablespace blocks. The tablespace block size in which the LOB is stored controls the value of `CHUNK`, which is a parameter of LOB storage. When you create a LOB column, you 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 equals 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 32 KB.

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 (TB) to 128 TB.

For example, suppose that your database block size is 32 KB and you create a tablespace with a nonstandard block size of 8 KB. Further suppose that you create a table with a LOB column and specify a `CHUNK` size of 16 KB (which is a multiple of the 8 KB tablespace block size). Then the maximum size of a LOB in this column is (4 gigabytes – 1) * 16 KB.

The maximum size of a `BFILE` is the maximum file size allowed in the operating system, or `UB8MAXVAL`, whichever is smaller.

Older LOB functions use `ub4` as the data types of some parameters, and the `ub4` data type can only hold up to 4 GB. The newer functions use parameters of 8-byte length, `oraub8`, which is a data type defined in `oratypes.h`. The data types `oraub8` and `orasb8` are mapped to appropriate 64-bit native data types 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 usable chunk size in bytes for `BLOB`s, `CLOB`s, and `NCLOB`s. 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 `BFILE`s larger than 4 gigabytes in any programmatic environment. An additional file size limit imposed by your operating system also applies to `BFILE`s.

This section includes the following topics:

- Functions to Use for the Increased LOB Sizes
- Compatibility and Migration
- OCILobGetChunkSize()
- OCILobGetStorageLimit()

## 9.6.1 Functions to Use for the Increased LOB Sizes

Eight functions with names that end in "2" and that use the data type `oraub8` in place of the data type `ub4` were introduced in Oracle Database 10*g* Release 1.

Other changes were 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` assumed either byte or char length for LOBs based on the locator type and character set. It was 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` parameters as replacement for the `amtp` parameter. The `char_amtp` parameter is preferred for `CLOB` and `NCLOB`, and the `byte_amtp` parameter is only considered as input if `char_amtp` is zero. On output for `CLOB` and `NCLOB`, both `byte_amtp` and `char_amtp` parameters are filled. For `BLOB` and `BFILE`, the `char_ampt` 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 the `byte_amtp` parameter or `char_amtp` parameter 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 the `byte_amtp` and `char_amtp` parameters must be updated after each call in polling mode.

Problem: While reading or writing data in streaming mode with callback, users must use the same buffer for each piece of data.

Solution: The callback function must have two new parameters to provide the buffer and the buffer length. Callback functions can set the 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
- OCILobRead2()
- OCILobWrite2()
- OCILobWriteAppend2()

## 9.6.2 Compatibility and Migration

Existing OCI programs can be enhanced to process larger amounts of LOB data that are greater than 4 GB.

Table 9-1 summarizes compatibility issues in this table, "old" refers to releases before Oracle Database 10*g* Release 1, and NA means not applicable.

**Table 9-1 LOB Functions Compatibility and Migration**

| LOB Function | Old Client/New or Old Server[1] | New Client/Old Server | New Client/New Server |
|---|---|---|---|
| OCILobArrayRead() | [2]NA | OK until piece size and offset are < 4 GB. | OK |
| OCILobArrayWrite() | NA | OK until piece size and offset are < 4 GB. | OK |
| OCILobCopy2() | NA | OK until LOB size, piece size (amount) and offset are < 4 GB. | OK |
| OCILobCopy() | OK; limit is 4 GB. | OK | OK; limit is 4 GB. |
| OCILobErase2() | NA | OK until piece size and offset are < 4 GB. | OK |
| OCILobErase() | OK; limit is 4 GB. | OK | OK; limit is 4 GB. |
| OCILobGetLength2() | NA | OK | OK |
| OCILobGetLength() | OK; limit is 4 GB. | OK | OK; `OCI_ERROR` if LOB size > 4 GB. |
| OCILobLoadFromFile2() | NA | OK until LOB size, piece size (amount), and offset are < 4 GB. | OK |
| OCILobLoadFromFile() | OK; limit is 4 GB. | OK | OK; limit is 4 GB. |
| OCILobRead2() | NA | OK until LOB size, piece size (amount), and offset are < 4 GB. | OK |
| OCILobRead() | OK; limit 4 GB.<br>With new server: `OCI_ERROR` is returned if you try to read any amount >= 4 GB from any offset < 4 GB. This is because when you read any amount >= 4 GB, that results in an overflow of returned value in *amtp, and so it is flagged as an error.<br>Note:<br>• If you read up to 4 GB – 1 from offset, that is not flagged as an error.<br>• When you use streaming mode with polling, no error is returned if no attempt is made to use piece size > 4 GB (you can read data > 4 GB in this case). | OK | OK.<br>`OCI_ERROR` is returned if you try to read any amount >= 4 GB from any offset < 4 GB. This is because when you read any amount >= 4 GB, that results in an overflow of returned value in *amtp, and so it is flagged as an error.<br>Note:<br>• If you read up to 4 GB – 1 from offset, that is not to be flagged as an error.<br>• When you use streaming mode with polling, no error is returned if no attempt is made to use piece size > 4 GB. |
| OCILobTrim2() | NA | OK | OK |
| OCILobTrim() | OK; limit 4 GB. | OK | OK; limit 4 GB. |

**Table 9-1 (Cont.) LOB Functions Compatibility and Migration**

| LOB Function | Old Client/New or Old Server[1] | New Client/Old Server | New Client/New Server |
|---|---|---|---|
| OCILobWrite2() | NA | OK until LOB size, piece size (amount) and offset are < 4 GB. | OK |
| OCILobWrite() | OK; limit 4 GB.<br>With new server:<br>`OCI_ERROR` is returned if you write any amount >= 4 GB (from any offset < 4 GB) because that results an in overflow of returned value in `*amtp`.<br>Note: Updating a LOB of 10 GB from any offset up to 4 GB –1 by up to 4 GB –1 amount of data is not flagged as an error. | OK | OK.<br>`OCI_ERROR` is returned if you write any amount >=4 GB (from any offset < 4 GB) because that results in an overflow of returned value in *amtp.<br>Note: Updating a LOB of 10 GB from any offset up to 4 GB –1 by up to 4 GB –1 amount of data is not flagged as an error. |
| OCILobWriteAppend2() | NA | OK until LOB size and piece size are <4 GB. | OK |
| OCILobWriteAppend() | OK; limit 4 GB.<br>With new server:<br>`OCI_ERROR` is returned if you append any amount >= 4 GB of data because that results in an overflow of returned value in `*amtp`. | OK | OK; limit 4 GB.<br>`OCI_ERROR` is returned if you append any amount >= 4 GB of data because that results in an overflow of returned value in `*amtp`. |
| OCILobGetStorageLimit() | NA | Error | OK |

[1] The term "old" refers to releases before Oracle Database 10g Release 1.

[2] NA means not applicable.

Use the functions that end in "2" when using the current server and current client. Mixing deprecated functions with functions that end in "2" can result in unexpected situations, such as data written using `OCILobWrite2()` being greater than 4 GB 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 gets an error message when the size crosses 4 GB and the deprecated functions are used. However, there is no issue if you use those deprecated functions for LOBs of size smaller than 4 GB.

## 9.7 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 `OCILobCopy2()`, `OCILobErase2()`, `OCILobLoadFromFile2()`, and `OCILobTrim2()`, the `amount` parameter is in characters for `CLOB`s and `NCLOB`s, regardless of the client-side character set.

These LOB operations refer to the amount of LOB data on the server. When the client-side character set is 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 `CLOB`s or `NCLOB`s and in bytes for `BLOB`s or `BFILE`s.

Exceptions to these general rules are noted in the description of the specific LOB call.

This section includes the following topics:

- About Improving LOB Read/Write Performance
- LOB Buffering Functions
- Functions for Opening and Closing LOBs
- LOB Read and Write Callbacks

**See Also:**

- LOB Functions
- Buffer Expansion During OCI Binding
- OCILobCopy2()
- OCILobErase2()
- OCILobLoadFromFile2()
- OCILobTrim2()

## 9.7.1 About Improving LOB Read/Write Performance

How to improve LOB Read/Write performance.

Here are some hints to improve performance:

- About Using Data Interface for LOBs
- About Using OCILobGetChunkSize()
- About Using OCILobWriteAppend2()
- About Using OCILobArrayRead() and OCILobArrayWrite()

### 9.7.1.1 About 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 that requires multiple round-trips.

**See Also:**

- About Binding LOB Data for usage and examples for both `INSERT` and `UPDATE` statements
- About Defining LOB Data for usage and examples of `SELECT` statements

### 9.7.1.2 About Using OCILobGetChunkSize()

`OCILobGetChunkSize()` returns the usable chunk size in bytes for `BLOB`s, `CLOB`s, and `NCLOB`s.

You can use the `OCILobGetChunkSize()` call to improve the performance of LOB read and write operations for BasicFile LOBs. When a read or write is done on BasicFile LOB data whose size is a multiple of the usable chunk size and the operation starts on a chunk boundary, performance is improved. There is no requirement for SecureFile LOBs to be written or read with `OCILobGetChunkSize()` alignment.

Calling `OCILobGetChunkSize()` returns the usable chunk size of the LOB, so that an application can batch a series of write operations for the entire chunk, rather than issuing multiple LOB write calls for the same chunk.

**See Also:**

- Options of SecureFiles LOBs
- OCILobGetChunkSize()

### 9.7.1.3 About 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 `OCILobGetLength2()` to determine the starting point for an `OCILobWrite2()` operation. `OCILobWriteAppend2()` does both steps.

**See Also:**

- OCILobWriteAppend2()
- OCILobGetLength2()
- OCILobWrite2()

### 9.7.1.4 About Using OCILobArrayRead() and OCILobArrayWrite()

You can improve performance by using by using `OCILobArrayRead()` to read LOB data for multiple LOB locators and `OCILobArrayWrite()` to write LOB data for multiple LOB locators.

These functions, which were introduced in Oracle Database 10g Release 2, reduce the number of round-trips for these operations.

> **See Also:**
>
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* sections "LOB Array Read" and "LOB Array Write" for more information and code examples that show how to use these functions with callback functions and in piecewise mode
> - OCILobArrayRead()
> - OCILobArrayWrite()

## 9.7.2 LOB Buffering Functions

These LOB functions are deprecated. Use the LOB prefetch feature in place of these LOB buffering functions.

OCI provides several calls for controlling LOB buffering for small reads and writes of internal LOB values:

- `OCILobEnableBuffering()`
- `OCILobDisableBuffering()`
- `OCILobFlushBuffer()`

These functions enable applications that are 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 SecureFiles and Large Objects Developer's Guide* for more information on LOB buffering, see the chapter about using LOB APIs.
- LOB Function Round-Trips for a list of the server round-trips required for each function
- Prefetching of LOB Data, Length, and Chunk Size
- OCILobEnableBuffering()
- OCILobDisableBuffering()
- OCILobFlushBuffer()

## 9.7.3 Functions for Opening and Closing LOBs

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 call can be performed on the same `BFILE` by using different locators.

If an application does not wrap LOB operations within 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 within 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 within 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 within 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, 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 is not updated. If this happens, rebuild your functional and domain indexes on the LOB column.

This section includes the following topic: Restrictions on Opening and Closing LOBs.

**See Also:**

- OCILobOpen()
- OCILobClose()
- OCILobIsOpen()

### 9.7.3.1 Restrictions on Opening and Closing LOBs

What are the restrictions on opening and closing LOBs.

The LOB opening and closing mechanism has the following restrictions:

- An application must close all previously opened LOBs before committing a transaction. Failing to do so results in an error. If a transaction is rolled back, all open LOBs are discarded along with the changes made. Because the LOBs are not closed, so the associated triggers are not fired.
- Although there is no limit to the number of open internal LOBs, there is a limit on the number of open files as determined by the `SESSION_MAX_OPEN_FILES` parameter. Assigning an already opened locator to another locator does not count as opening a new LOB.
- 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.
- 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:**

- `SESSION_MAX_OPEN_FILES` parameter in *Oracle Database Reference*
- OCI Demonstration Programs for examples of the use of the OCILobOpen() and OCILobClose() calls in the online demonstration programs
- Table C-2

## 9.7.4 LOB Read and Write Callbacks

OCI supports read and write callback functions.

The following sections describe the use of callbacks in more detail:

- Callback Interface for Streaming
- Reading LOBs by Using Callbacks
- Writing LOBs by Using Callbacks

### 9.7.4.1 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 LOBs.

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 they are required.

**See Also:**

- OCILobRead2()
- OCILobWriteAppend2()
- OCILobWrite2()

### 9.7.4.2 Reading LOBs by Using Callbacks

The user-defined read callback function is registered through the `OCILobRead2()` function.

The callback function should have the following prototype:

```
CallbackFunctionName ( void *ctxp, CONST void *bufp, oraub8 len, ub1 piece,
                       void **changed_bufpp, oraub8 *changed_lenp);
```

The first parameter, `ctxp`, is the context of the callback that is passed to OCI in the `OCILobRead2()` function call. When the callback function is called, the information provided by you in `ctxp` is passed back to you (OCI does not use this information on the way IN). The `bufp` parameter in `OCILobRead2()`) 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. The `changed_bufpp` parameter should point to the address of the changed buffer and the `changed_lenp` parameter should point to the length of the changed buffer. The `changed_bufpp` and `changed_lenp` parameters need not be used inside the callback function if the application does not change the buffer dynamically.

Example 9-1 shows a code fragment that 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.

### Example 9-1 Implementing Read Callback Functions Using OCILobRead2()

```
...
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,
  (void *) bufp, (oraub8) MAXBUFLEN, (void *) 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)
void       *ctxp;
CONST void *bufxp;
oraub8      len;
ub1         piece;
void       **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 must 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 must be changed dynamically --*/
          break;
      default:
          (void) printf("callback read error: unknown piece = %d.\n", piece);
           return OCI_ERROR;
     }
    return OCI_CONTINUE;
}
```

> **See Also:**
>
> OCILobRead2()

## 9.7.4.3 Writing LOBs by 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 ( void *ctxp, void *bufp, oraub8 *lenp, ub1 *piecep,
                        void **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 OCI (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 `OCILobWrite2()`.

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 `lenp`. You also indicate whether it is the next (`OCI_NEXT_PIECE`) or the last (`OCI_LAST_PIECE`) piece using the `piecep` parameter. You must ensure that the storage pointer that is provided by the application 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. The `changed_bufpp` parameter should point to the address of the changed buffer and the `changed_lenp` parameter should point to the length of the changed buffer. The `changed_bufpp` and `changed_lenp` parameters need not be used inside the callback function if the application does not change the buffer dynamically.

Example 9-2 shows a code fragment that 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.

**Example 9-2 Implementing Write Callback Functions Using OCILobWrite2()**

```
...

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,
    (void*)bufp, (ub4)nbytes, OCI_FIRST_PIECE, (void *)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)
void    *ctxp;
void    *bufxp;
oraub8  *lenp;
ub1       *piecep;
void  **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 must be changed dynamically --*/
    if (this is the last data buffer)
        *piecep = OCI_LAST_PIECE;
    else
        *piecep = OCI_NEXT_PIECE;
    return OCI_CONTINUE;
}
```

**See Also:**

OCILobWrite2()

## 9.8 Temporary LOB Support

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 that 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` operation. If the temporary LOB is not freed explicitly, it is not freed until the end of its specified `duration`. Unless you have another temporary locator pointing to the same LOB, you no longer have a locator pointing to the temporary LOB, because the original locator was overwritten by the `SELECT...INTO` operation.

This section includes the following topics:

- Creating and Freeing Temporary LOBs
- Temporary LOB Durations
- About Freeing Temporary LOBs
- Take Care When Assigning Pointers
- Temporary LOB Example

**See Also:**

- OCILobCreateTemporary()
- OCILobFreeTemporary()
- OCILobIsTemporary()

## 9.8.1 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 the temporary LOB is read into the server's buffer cache.

To make a temporary LOB permanent, use `OCILobCopy2()` 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 that are used for standard LOBs.

**Note:**

The most efficient way to insert an empty LOB is to bind a temporary LOB with no value assigned to it. This uses less resources than the following method.

```
INSERT INTO tab1 VALUES(EMPTY_CLOB())
```

**See Also:**

- OCILobCreateTemporary()
- OCILobFreeTemporary()
- OCILobCopy2()

## 9.8.2 Temporary LOB Durations

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:

- Call, `OCI_DURATION_CALL`, only on the server side
- Session, `OCI_DURATION_SESSION`

The session duration expires when the containing session or connection ends. The call duration expires at the end of the current OCI call.

When you run in object mode, 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.

**See Also:**

- OCIDurationBegin()
- OCIDurationEnd()
- OCIDescriptorFree()

## 9.8.3 About Freeing Temporary LOBs

Any time that your OCI program obtains a LOB locator from SQL or PL/SQL, use the `OCILobIsTemporary()` function to check that the locator is temporary.

If it is, then free the locator when your application is finished with it by using the `OCILobFreeTemporary()` call. The locator can be from a define during a select or an out bind. A temporary LOB duration is always upgraded to a session duration when it is shipped to the client side. The application must 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()
- OCILobFreeTemporary()

## 9.8.4 Take Care When Assigning Pointers

Special care must 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, you must ensure that any temporary LOB in the target LOB locator is freed by `OCILobFreeTemporary()`. 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, you must free any temporary LOB in the existing out-bind LOB locator buffer by using the `OCILobFreeTemporary()` call.

**See Also:**

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information in the section about temporary LOB performance guidelines
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for a discussion of optimal performance of temporary LOBs
- OCILobAssign()
- OCILobLocatorAssign()
- OCILobFreeTemporary()
- OCILobLocatorAssign()

## 9.8.5 Temporary LOB Example

Shows how temporary LOBs can be used.

Example 9-3 shows how temporary LOBs can be used.

**Example 9-3 Using Temporary LOBs**

```
#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 that 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((void*)envhp, (void **)&tblob,
                          (ub4)OCI_DTYPE_LOB, (size_t)0, (void**)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 (OCIStmtPrepare2(svchp, stmthp, errhp, sqlstmt, (ub4) strlen((char *)sqlstmt),
                     NULL, 0, (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, (void *) &lob_loc, (sb4)0,
             (ub2) SQLT_BLOB, (void *)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 Log on */
  OCIEnvCreate(&envhp, OCI_DEFAULT, (void *)0, 0, 0, 0,
        (size_t)0, (void *)0);
  (void) OCIHandleAlloc( (void *) envhp, (void **) &errhp, OCI_HTYPE_ERROR,
                          (size_t) 0, (void **) 0);
  /* server contexts */
  (void) OCIHandleAlloc( (void *) envhp, (void **) &srvhp, OCI_HTYPE_SERVER,
                         (size_t) 0, (void **) 0);
   /* service context */
  (void) OCIHandleAlloc( (void *) envhp, (void **) &svchp, OCI_HTYPE_SVCCTX,
                         (size_t) 0, (void **) 0);
    /* attach to Oracle Database */
  (void) OCIServerAttach( srvhp, errhp, (text *)"", strlen(""), 0);
    /* set attribute server context in the service context */
  (void) OCIAttrSet ((void *) svchp, OCI_HTYPE_SVCCTX,
                     (void *)srvhp, (ub4) 0,
                     OCI_ATTR_SERVER, (OCIError *) errhp);
   (void) OCIHandleAlloc((void *) envhp,
                        (void **)&authp, (ub4) OCI_HTYPE_SESSION,
                        (size_t) 0, (void **) 0);
   (void) OCIAttrSet((void *) authp, (ub4) OCI_HTYPE_SESSION,
                    (void *) "scott", (ub4)5,
                    (ub4) OCI_ATTR_USERNAME, errhp);
  (void) OCIAttrSet((void *) authp, (ub4) OCI_HTYPE_SESSION,
                    (void *) "password", (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((void *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                    (void *) authp, (ub4) 0,
                    (ub4) OCI_ATTR_SESSION, errhp);
  /* ------- Done logging in ----------------------------------*/
   /* allocate a statement handle */
  checkerr(errhp, OCIHandleAlloc( (void *) envhp, (void **) &stmthp,
                             OCI_HTYPE_STMT, (size_t) 0, (void **) 0));
  checkerr(errhp, OCIDescriptorAlloc((void *)envhp, (void **)&lob_loc,
                   (ub4) OCI_DTYPE_LOB, (size_t) 0, (void **) 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((void *)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;
    }
  }
```

## 9.9 Prefetching of LOB Data, Length, and Chunk Size

To improve OCI access of smaller LOBs, LOB data can be prefetched and cached while also fetching the locator.

This applies to internal LOBs, temporary LOBs, and `BFILE`s. Take the following steps to prepare your application:

1. Set the `OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE` attribute for the session handle. The value of this attribute indicates the default prefetch data size for a LOB locator. This attribute value enables prefetching for all the LOB locators fetched in the session. The default value for this attribute is zero (no prefetch of LOB data). This option relieves the application developer from setting the prefetch LOB size for each define handle. You can either set this attribute or set (in Step 3) `OCI_ATTR_LOBPREFETCH_SIZE`.
2. Perform the prepare and define steps for the statement to be executed.
3. You can override the default prefetch size, if required, for the LOB locators to be fetched, by setting `OCI_ATTR_LOBPREFETCH_SIZE` attribute for the define handle. This optional attribute provides control of the prefetch size for the locators fetched from a particular column.
4. Set the `OCI_ATTR_LOBPREFETCH_LENGTH` attribute to the prefetch LOB length and chunk size. This is mandatory to be set to `TRUE` for the attribute (`OCI_ATTR_LOBPREFETCH_SIZE`) described in item 1 to work.
5. Execute the statement.
6. Call OCILobRead2() or OCILobArrayRead() with individual LOB locators; OCI takes the data from the prefetch buffer, does the necessary character conversion, and copies the data into the LOB read buffer (no change in LOB semantic). If the

data requested is bigger than the prefetch buffer, then it will require additional round-trips.

7. Call OCILobGetLength2() and OCILobGetChunkSize() to obtain the length and chunk size without making round-trips to the server.

**Example 9-4    Prefetching of LOB Data, Length, and Chunk Size**

```
...
ub4 default_lobprefetch_size = 2000;                  /* Set default size to 2K */
...
/* set LOB prefetch attribute to session */
OCIAttrSet (sesshp, (ub4) OCI_HTYPE_SESSION,
            (void *)&default_lobprefetch_size,               /* attribute value */
            0,                     /* attribute size; not required to specify; */
            (ub4) OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE,
            errhp);
...
/* select statement */
char *stmt = "SELECT lob1 FROM lob_table";
...
/* declare and allocate LOB locator */
OCILobLocator * lob_locator;
lob_locator = OCIDescriptorAlloc(..., OCI_DTYPE_LOB, ...);

OCIDefineByPos(..., 1, (void *) &lob_locator, ..., SQLT_CLOB, ...);
...
/* Override the default prefetch size to 4KB */
ub4 prefetch_size = 4000;
OCIAttrSet (defhp,  OCI_HTYPE_DEFINE,
            (void *) &prefetch_size                             /* attr value */,
            0                  /* restricting prefetch size to be ub4 max val */,
            OCI_ATTR_LOBPREFETCH_SIZE                            /* attr type */,
            errhp);
...
/* Set prefetch length attribute */
boolean prefetch_length = TRUE;
OCIAttrSet( defhp,  OCI_HTYPE_DEFINE,
            (dvoid *) &prefetch_length  /* attr value */,
            0,
            OCI_ATTR_LOBPREFETCH_LENGTH  /* attr type */,
            errhp );
...
/* execute the statement. 4KB of data for the LOB is read and
 * cached in descriptor cache buffer.
 */
OCIStmtExecute (svchp, stmthp, errhp,
                1,               /* iters */
                0,               /* row offset */
                NULL,            /* snapshot IN */
                NULL,            /* snapshot out */
                OCI_DEFAULT);    /* mode */
...
oraub8 char_amtp = 4000;
oraub8 lob_len;
ub4 chunk_size;

/* LOB chunk size, length, and data are read from cache. No round-trip. */

OCILobGetChunkSize (svchp, errhp, lob_locator, &chunk_size);

OCILobGetLength2(svchp, errhp, lob_locator, &lob_len );
```

```
OCILobRead2(svchp, errhp, lob_locator, NULL, &char_amtp, ...);
...
```

Note that the prefetch size is in number of bytes for `BLOB`s and `BFILE`s and in number of characters for `CLOB`s.

Example 9-4 shows a code fragment illustrating these steps.

**Prefetch cache allocation:** The prefetch cache buffer for a descriptor is allocated while fetching a LOB locator. The allocated buffer size is determined by the `OCI_ATTR_LOBPREFETCH_SIZE` attribute for the define handle; the default value of this attribute is indicated by the `OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE` attribute value of the session handle. If the cache buffer is already allocated, then it is resized if required.

For the following two LOB APIs, if the source locator has cached data, then the destination locator cache is allocated or resized and cached data is copied from source to destination.

- OCILobAssign()
- OCILobLocatorAssign()

Once allocated, the cache buffer memory for a descriptor is released when the descriptor itself is freed.

**Prefetch cache invalidation:** The cache for a descriptor gets invalidated when LOB data is updated using the locator. Meaning the cache is no longer used for reading data and the next OCILobRead2() call on the locator makes a round-trip.

The following LOB APIs invalidate the prefetch cache for the descriptor used:

- OCILobErase() (deprecated)
- OCILobErase2()
- OCILobTrim() (deprecated)
- OCILobTrim2()
- OCILobWrite() (deprecated)
- OCILobWrite2()
- OCILobWriteAppend() (deprecated)
- OCILobWriteAppend2()
- OCILobArrayWrite()

The following LOB APIs invalidate the cache for the destination LOB locator:

- OCILobAppend()
- OCILobCopy() (deprecated)
- OCILobCopy2()
- OCILobLoadFromFile() (deprecated)
- OCILobLoadFromFile2()

**Performance Tuning:** The prefetch buffer size must be decided upon based on average LOB size and client-side memory. If a large amount of data is prefetched, you must ensure the memory availability. Performance gain may not be significant for prefetching large LOBs, because the cost of fetching data is much higher compared to the cost of a round-trip to the server.

You must have a fair idea of the LOB data size to be able to make best use of this LOB prefetch feature. Because the parameters are part of application design, the application must be rebuilt if any parameter value must be modified.

**Upgrading:** LOB prefetching cannot be used against a pre-11.1 release server or in a pre-11.1 client against an 11.1 or later server. When you use a pre-11.1 server with an 11.1 or later client, OCIAttrSet() returns an error or an error-with-information saying that "server does not support this functionality."

> **See Also:**
>
> - OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE
> - OCI_ATTR_LOBPREFETCH_LENGTH
> - OCI_ATTR_LOBPREFETCH_SIZE

## 9.10 Options of SecureFiles LOBs

For SecureFiles (LOBs with the `STORE AS SECUREFILE` option, which were introduced in Oracle Database 11*g* Release 1) you can specify the SQL parameter `DEDUPLICATE` in `CREATE TABLE` and `ALTER TABLE` statements.

This parameter value enables you to specify that LOB data that is identical in two or more rows in a LOB column shares the same data blocks, thus saving disk space. `KEEP_DUPLICATES` turns off this capability. The following options are also used with `SECUREFILE`:

The parameter `COMPRESS` turns on LOB compression. `NOCOMPRESS` turns LOB compression off.

The parameter `ENCRYPT` turns on LOB encryption and optionally selects an encryption algorithm. `NOENCRYPT` turns off LOB encryption. Each LOB column can have its own encryption specification, independent of the encryption of other LOB or non-LOB columns. Valid algorithms are `3DES168`, `AES128`, `AES192`, and `AES256`.

The LOBs paradigm used before release 11.1 is the default. This default LOBs paradigm is also now explicitly set by the option `STORE AS BASICFILE`.

The following OCI functions are used with the `SECUREFILE` features:

- `OCILobGetOptions()`
- `OCILobSetOptions()`
- `OCILobGetContentType()`
- `OCILobSetContentType()`

**See Also:**

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for complete details of relevant SQL functions and cross-references to PL/SQL packages and information about migrating to SecureFiles
- OCILobGetOptions()
- OCILobSetOptions()
- OCILobGetContentType()
- OCILobSetContentType()

# 10

# Managing Scalable Platforms

This chapter describes how OCI manages transaction operations.

This chapter contains these topics:

- OCI Support for Transactions
- Levels of Transactional Complexity
- Password and Session Management
- Middle-Tier Applications in OCI
- Externally Initialized Context in OCI
- Client Application Context
- Using Edition-Based Redefinition
- OCI Security Enhancements
- Overview of OCI Multithreaded Development
- OCIThread Package

## 10.1 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 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
- OCITransStart()
- OCITransDetach()
- OCITransCommit()
- OCITransRollback()
- OCITransPrepare()
- OCITransMultiPrepare()
- OCITransForget()

# 10.2 Levels of Transactional Complexity

OCI supports several levels of transaction complexity.

These levels of transaction complexity include:

- Simple Local Transactions
- Serializable or Read-Only Local Transactions
- Global Transactions

This section includes the following additional topics:

- Transaction Examples
- Initialization Parameters

## 10.2.1 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() for sample code showing the use of simple local transactions
- OCITransRollback()

## 10.2.2 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.

**See Also:**

OCITransStart()

## 10.2.3 Global Transactions

Global transactions are necessary only in more sophisticated transaction-processing applications.

This section includes the following topics:

- Transaction Identifiers
- Attribute OCI_ATTR_TRANS_NAME
- Transaction Branches
- Branch States
- Detaching and Resuming Branches
- About Setting the Client Database Name
- One-Phase Commit Versus Two-Phase Commit
- Preparing Multiple Branches in a Single Message

### 10.2.3.1 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 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.

Table 10-1 provides the structure for one possible `XID` of 1234.

**Table 10-1    Global Transaction Identifier**

| Component | Value |
|---|---|
| `gtrid` | `12` |
| `bqual` | `34` |
| `gtrid+bqual=XID` | `1234` |

The transaction identifier used by OCI transaction calls is set in the `OCI_ATTR_XID` attribute of the transaction handle, by using `OCIAttrSet()`. Alternately, the transaction can be identified by a name set in the `OCI_ATTR_TRANS_NAME` attribute.

> **See Also:**
>
> OCIAttrSet()

### 10.2.3.2 Attribute OCI_ATTR_TRANS_NAME

When this attribute is set 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 this attribute is retrieved from a transaction handle, the returned transaction name is the global transaction identifier. The size is the length of the global transaction identifier.

### 10.2.3.3 Transaction Branches

Within a single global transaction, Oracle Database supports both tightly coupled and loosely coupled relationships between a pair of branches.

- Tightly coupled branches share the same local transaction. 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. 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` values to specify the type of coupling.

A session corresponds to a user session, created with `OCISessionBegin()`.

Figure 10-1 illustrates tightly coupled branches within an application. The `XID`s of the two branches (B1 and B2) share the same `gtrid`, because they are operating on the same transaction (T), but they have a different `bqual`, because they are on separate branches.

**Figure 10-1 Multiple Tightly Coupled Branches**

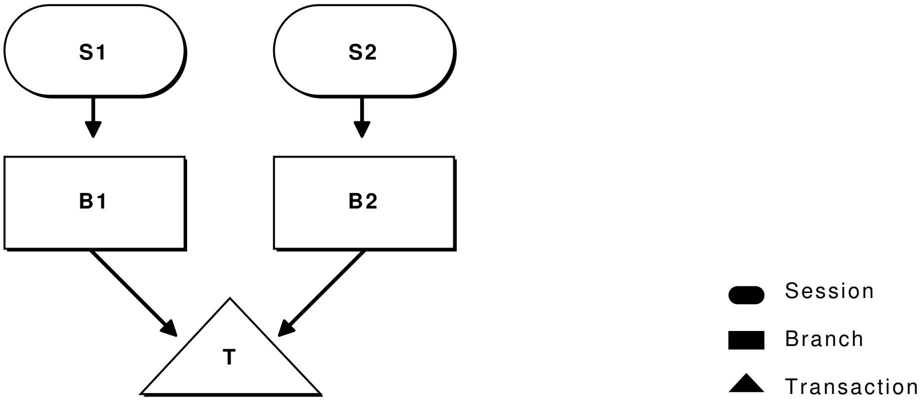
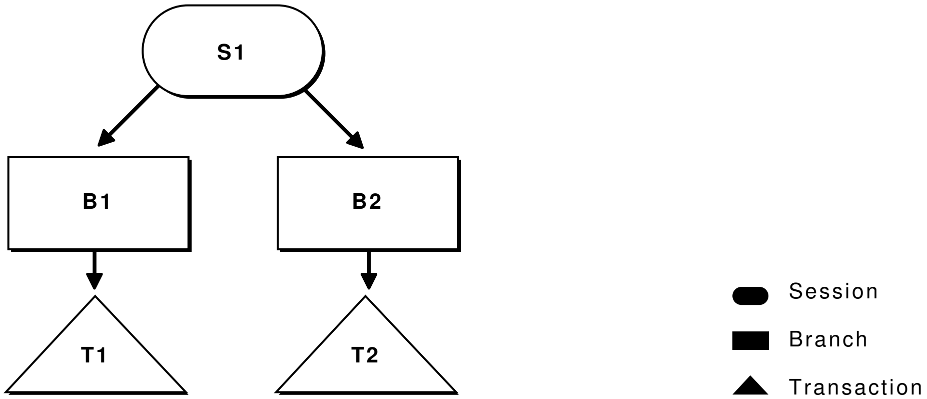

S1
S2
B1
B2
T
Session
Branch
Transaction

Figure 10-2 illustrates how a single session operates on different branches. The `gtrid` components of the `XIDs` are different, because they represent separate global transactions.

**Figure 10-2 Session Operating on Multiple Branches**


S1
B1
B2
T1
T2
Session
Branch
Transaction

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() for sample code demonstrating this scenario
- OCISessionBegin()

### 10.2.3.4 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.

### 10.2.3.5 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.

To resume a branch, the application 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 waits 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, if that process has the same authorization as the one that detached the transaction.

**See Also:**

- OCITransDetach()
- OCITransStart()

### 10.2.3.6 About Setting the 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.

> **Note:**
>
> An OCI application sets these attributes, by using `OCIAttrSet()` before logging on and using global transactions.

> **See Also:**
>
> OCIAttrSet()

### 10.2.3.7 One-Phase Commit Versus Two-Phase Commit

Global transactions can 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 Oracle databases. In this case, a two-phase commit is necessary. A two-phase commit operation consists of these steps:

1. Prepare - The application issues an `OCITransPrepare()` call against each transaction. Each transaction returns a value indicating whether or not it can 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 an `OCITransCommit()` call to each transaction. The `flags` parameter of the commit call must be explicitly set to `OCI_TRANS_TWOPHASE` for the appropriate behavior, because the default for this call is for one-phase commit.

   > **Note:**
   >
   > The `OCITransPrepare()` call can also return `OCI_SUCCESS_WITH_INFO` if a transaction must indicate that it is read-only. Thus a commit is neither appropriate nor necessary.

An additional call, `OCITransForget()`, causes a database to "forget" a completed transaction. This call is for situations in which a problem has occurred that requires that a two-phase commit be terminated. When an Oracle database receives an `OCITransForget()` call, it removes all information about the transaction.

**See Also:**

- OCITransCommit()
- OCITransPrepare()
- OCITransForget()

### 10.2.3.8 Preparing Multiple Branches in a Single Message

Sometimes when multiple applications use 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. The traditional `OCITransPrepare()` call is inefficient for this task as each branch must be individually prepared. The `OCITransMultiPrepare()` call, prepares multiple branches involved in the same global transaction in one round-trip. This call is more efficient and can greatly reduce the number of messages sent from the client to the server.

**See Also:**

- OCITransPrepare()
- OCITransMultiPrepare()

## 10.2.4 Transaction Examples

Shows how to use the transaction OCI calls.

Table 10-1 through Table 10-5 illustrate how to use the transaction OCI calls.

They show a series of OCI calls and other actions, along with their resulting behavior. For simplicity, not all parameters to these calls are listed; rather, it is the flow of calls that 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.

## 10.2.5 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, which controls the number of connections from a session and is not applicable to applications that use global transactions.

This section includes the following topics:

- Showing Update Successfully, One-Phase Commit
- Showing Starting a Transaction, Detach, Resume, Prepare, Two-Phase Commit
- Showing a Read-Only Update Fails
- Showing Starting a Read-Only Transaction, Select, and Commit

### 10.2.5.1 Showing Update Successfully, One-Phase Commit

Lists the steps for a one-phase commit operation.

Table 10-2 lists the steps for a one-phase commit operation.

**Table 10-2 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. |

**See Also:**

- OCITransStart()
- OCITransCommit()

### 10.2.5.2 Showing Starting a Transaction, Detach, Resume, Prepare, Two-Phase Commit

Lists the steps for a two-phase commit operation.

Table 10-3 lists the steps for a two-phase commit operation.

**Table 10-3 Two-Phase Commit**

| Step | OCI Action | XID | Flags | Result |
|---|---|---|---|---|
| 1 | `OCITransStart()` | 1234 | `OCI_TRANS_NEW` | Starts new read-only transaction |
| 2 | `SQL UPDATE` | - | - | Updates rows |
| 3 | `OCITransDetach()` | - | - | Transaction is detached. |
| 4 | `OCITransStart()` | 1234 | `OCI_TRANS_RESUME` | Transaction is resumed. |
| 5 | `SQL UPDATE` | - | - | - |
| 6 | `OCITransPrepare()` | - | - | Transaction is 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.

**See Also:**

- OCITransStart()
- OCITransDetach()
- OCITransPrepare()
- OCITransCommit()

## 10.2.5.3 Showing a Read-Only Update Fails

Lsts the steps in a failed read-only update operation.

Table 10-4 lists the steps in a failed read-only update operation.

**Table 10-4 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 the transaction is read-only. |
| 3 | `OCITransCommit()` | - | - | Commit has no effect. |

**See Also:**

- OCITransStart()
- OCITransCommit()

### 10.2.5.4 Showing Starting a Read-Only Transaction, Select, and Commit

Lists the steps for a read-only transaction.

Table 10-5 lists the steps for a read-only transaction.

**Table 10-5 Read-Only Transaction**

| Step | OCI Action | XID | Flags | Result |
|---|---|---|---|---|
| 1 | `OCITransStart()` | 1234 | `OCI_TRANS_NEW` \|<br>`OCI_TRANS_READONLY` | Starts new read-only transaction |
| 2 | `SQL SELECT` | - | - | Queries the database |
| 3 | `OCITransCommit()` | - | - | No effect — transaction is read-only, no changes made |

**See Also:**

- OCITransStart()
- OCITransCommit()

# 10.3 Password and Session Management

OCI can authenticate and maintain multiple users.

This section includes the following topics:

- OCI Authentication Management
- OCI Password Management
- Secure External Password Store
- OCI Session Management

## 10.3.1 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 database 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 can call `OCISessionBegin()` again to initialize another user session handle with different or the same credentials and different or the same operation modes. For an application to authenticate a user in `OCI_MIGRATE` mode, the service handle must already be associated with a nonmigratable user handle. The `userid` of that user handle becomes the ownership ID of the migratable user session. Every migratable session must have a nonmigratable parent session.

- If `OCI_MIGRATE` mode is not specified, then the user session context can be used only with the server handle specified 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 can 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 nonmigratable session currently connected to that same server.

`OCI_SYSDBA`, `OCI_SYSOPER`, `OCI_SYSBKP`, `OCI_SYSDGD`, `OCI_SYSKMT`, `OCI_SYSRAC`, `OCI_SYSASM`, and `OCI_PRELIM_AUTH` settings can only be used with a primary user session context. For `OCI_SYSASM`, the SYSASM connection works only to ASM instances (not database instances).

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 must 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.
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 reused 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`, these are ignored if `OCI_CRED_EXT` is used.

> **See Also:**
>
> - User Session Handle Attributes for more information about `OCI_ATTR_MIGSESSION`
> - OCISessionBegin()
> - OCIServerAttach()
> - OCIAttrSet()
> - OCISessionEnd()

## 10.3.2 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, the call 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.

> **See Also:**
>
> - OCIPasswordChange()
> - OCISessionBegin()

## 10.3.3 Secure External Password Store

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 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 Administrator's Guide* for information about configuring your client to use the secure external password store

## 10.3.4 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 Database 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 by using the `OCIAttrSet()` call, as shown in Example 10-1.

The server group name is an alphanumeric string not exceeding 30 characters. This attribute can only be set *after* calling `OCIServerAttach()`. `OCI_ATTR_SERVER_GROUP` attribute must be set in the server context before creating the first nonmigratable 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.

All migratable sessions created on servers within a server group can only migrate to other servers in the same server group. Servers that terminate are removed from the server group. New servers can 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 nonmigratable session created in a nondefault server group becomes the owner of the server group. All subsequent nonmigratable 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`.

**Example 10-1 Defining the OCI_ATTR_SERVER_GROUP Attribute to Pass the Server Group Name**

```
OCIAttrSet ((void *) srvhp, (ub4) OCI_HTYPE_SERVER, (void *) group_name,
            (ub4) strlen ((CONST char *) group_name),
            (ub4) OCI_ATTR_SERVER_GROUP, errhp);
```

**See Also:**

- Server Handle Attributes for more information about `OCI_ATTR_SERVER_GROUP` attribute
- OCIAttrSet()
- OCIServerAttach()

## 10.4 Middle-Tier Applications in OCI

A middle-tier application receives requests from browser clients.

The application 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 they 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 can 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 after 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` sets the database user name of the client.
- `OCI_ATTR_PROXY_CREDENTIALS` indicates the authenticated application making the proxy request.

To specify a list of roles activated after the application server 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 before the `OCISessionBegin()` call. Then the role is established and proxy capability is 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 fails.

This section includes the following topic: OCI Attributes for Middle-Tier Applications.

> **See Also:**
>
> - OCIAttrSet()
> - OCISessionBegin()

## 10.4.1 OCI Attributes for Middle-Tier Applications

The following attributes enable 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.

These attributes include:

- OCI_CRED_PROXY
- OCI_ATTR_PROXY_CREDENTIALS
- OCI_ATTR_DISTINGUISHED_NAME
- OCI_ATTR_CERTIFICATE
- OCI_ATTR_INITIAL_CLIENT_ROLES
- OCI_ATTR_CLIENT_IDENTIFIER
- OCI_ATTR_PASSWORD

### 10.4.1.1 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, use `OCI_CRED_PROXY` as the value passed in the `credt` parameter of `OCISessionBegin()` rather than `OCI_CRED_RDBMS` (database user name and password required) or `OCI_CRED_EXT` (externally provided credentials).

> **See Also:**
>
> OCISessionBegin()

### 10.4.1.2 OCI_ATTR_PROXY_CREDENTIALS

Use the `OCI_ATTR_PROXY_CREDENTIALS` attribute to specify the credentials of the application server in client authentication.

You can code the following declarations and `OCIAttrSet()` call, as shown in Example 10-2.

**Example 10-2 Defining the OCI_ATTR_PROXY_CREDENTIALS Attribute to Specify the Credentials of the Application Server for Client Authentication**

```
OCISession *session_handle;
OCISvcCtx  *application_server_session_handle;
OCIError   *error_handle;
...
OCIAttrSet((void *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (void *)application_server_session_handle, (ub4) 0,
            OCI_ATTR_PROXY_CREDENTIALS, error_handle);
```

**See Also:**

OCIAttrSet()

### 10.4.1.3 OCI_ATTR_DISTINGUISHED_NAME

Use `OCI_ATTR_DISTINGUISHED_NAME` to pass the distinguished name of the client.

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`, as shown in Example 10-3.

**Example 10-3 Defining the OCI_ATTR_DISTINGUISHED_NAME Attribute to Pass the Distinguished Name of the Client**

```
/* Declarations */
...
OCIAttrSet((void *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (void *)distinguished_name, (ub4) 0,
           OCI_ATTR_DISTINGUISHED_NAME, error_handle);
```

**See Also:**

OCIAttrSet()

### 10.4.1.4 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 the 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`, as shown in Example 10-4.

**Example 10-4 Defining the OCI_ATTR_CERTIFICATE Attribute to Pass the Entire X.509 Certificate**

```
OCIAttrSet((void *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (void *)certificate, ub4 certificate_length,
           OCI_ATTR_CERTIFICATE, error_handle);
```

> **See Also:**
>
> OCIAttrSet()

## 10.4.1.5 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 database.

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, as shown in Example 10-5.

**Example 10-5 Defining the OCI_ATTR_INITIAL_CLIENT_ROLES Attribute to Pass the Client Roles**

```
OCIAttrSet((void *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (void *)role_array, (ub4) number_of_strings,
           OCI_ATTR_INITIAL_CLIENT_ROLES, error_handle);
```

> **See Also:**
>
> OCIAttrSet()

## 10.4.1.6 OCI_ATTR_CLIENT_IDENTIFIER

Use `OCI_ATTR_CLIENT_IDENTIFIER` on the middle tier to track end-user identity.

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 the identity of these 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()`, as shown in Example 10-6. On the next request to the server, the information is propagated and stored in the server session.

The `CLIENT_IDENTIFIER`, which can also be set using the `DBMS_SESSION.SET_IDENTIFIER` procedure, is automatically truncated if the supplied input is more than 64 bytes because the maximum size for the client id is 64 bytes.

`OCI_ATTR_CLIENT_IDENTIFIER` can also be used in conjunction with the global application context to restrict availability of the context to the selected identity of these users.

When a client has multiple sessions, execute `OCIAttrSet()` for each session using the same client identifier. `OCIAttrSet()` must be executed manually for sessions that are reestablished through transparent application failover (TAF).

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;
```

**Example 10-6 Defining the OCI_ATTR_CLIENT_IDENTIFIER Attribute to Pass the End-User Identity**

```
OCIAttrSet((void *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (void *)"janedoe", (ub4)strlen("janedoe"),
           OCI_ATTR_CLIENT_IDENTIFIER, error_handle);
```

**See Also:**

- *Oracle Database Security Guide* the section about preserving user identity in multi-tiered environments"
- Transparent Application Failover in OCI
- OCIAttrSet()

## 10.4.1.7 OCI_ATTR_PASSWORD

To use the password of the client for validation on the middle-tier, the application server can use the existing attribute `OCI_ATTR_PASSWORD` with the client’s authentication data.

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. Although it appears that this is the same as a client/server connection, the client does not have to have Oracle Database 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`, as shown in Example 10-7.

Example 10-8 shows OCI attributes that enable you to specify the external name and initial privileges of a client. These credentials are used by OCI applications as alternative means of identifying or authenticating the client.

**Example 10-7 Defining the OCI_ATTR_PASSWORD Attribute to Pass the Password for Validation**

```
OCIAttrSet((void *)session_handle, (ub4) OCI_HTYPE_SESSION, (void *)password,
           (ub4)0, OCI_ATTR_PASSWORD, error_handle);
```

### Example 10-8 OCI Attributes That Let You Specify the External Name and Initial Privileges of a Client

```
...
*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, (void *)0,
                     (void * (*)(void *, size_t)) 0,
                     (void * (*)(void *, void *, size_t))0,
                     (void (*)(void *, void *)) 0 );
(void) OCIEnvInit( (OCIEnv **) &environment_handle, OCI_DEFAULT, (size_t) 0,
                  (void **) 0 );
(void) OCIHandleAlloc( (void *) environment_handle, (void **) &error_handle,
     OCI_HTYPE_ERROR, (size_t) 0, (void **) 0);
/*
** Allocate and initialize the server and service contexts used by the
** application server.
*/

(void) OCIHandleAlloc( (void *) environment_handle,
     (void **)&data_server_handle, OCI_HTYPE_SERVER, (size_t) 0, (void **) 0);
(void) OCIHandleAlloc( (void *) environment_handle, (void **)
     &application_server_service_handle, OCI_HTYPE_SVCCTX, (size_t) 0,
     (void **) 0);
(void) OCIAttrSet((void *) application_server_service_handle,
     OCI_HTYPE_SVCCTX, (void *) data_server_handle, (ub4) 0, OCI_ATTR_SERVER,
     error_handle);
/*
** Authenticate the application server. In this case, external authentication is
** being used.
*/

(void) OCIHandleAlloc((void *) environment_handle,
     (void **)&application_server_session_handle, (ub4) OCI_HTYPE_SESSION,
     (size_t) 0, (void **) 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((void *) environment_handle,
     (void **)&first_client_session_handle, (ub4) OCI_HTYPE_SESSION,
     (size_t) 0,(void **) 0);
(void) OCIAttrSet((void *) first_client_session_handle,
     (ub4) OCI_HTYPE_SESSION, (void *) "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((void *) first_client_session_handle,
     (ub4) OCI_HTYPE_SESSION, (void *) application_server_session_handle,
     (ub4) 0, OCI_ATTR_PROXY_CREDENTIALS, error_handle);
(void) OCIAttrSet((void *) first_client_session_handle,
     (ub4) OCI_HTYPE_SESSION, (void *) "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((void *) 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.
** This code is unchanged from the current way of doing session switching.
*/

(void) OCIHandleAlloc((void *) environment_handle,
     (void **)&second_client_session_handle, (ub4) OCI_HTYPE_SESSION,
     (size_t) 0, (void **) 0);
(void) OCIAttrSet((void *) second_client_session_handle,
     (ub4) OCI_HTYPE_SESSION, (void *) "mutt", (ub4) strlen("mutt"),
     OCI_ATTR_USERNAME, error_handle);
(void) OCIAttrSet((void *) second_client_session_handle,
     (ub4) OCI_HTYPE_SESSION, (void *) application_server_session_handle,
     (ub4) 0, OCI_ATTR_PROXY_CREDENTIALS, error_handle);
(void) OCIAttrSet((void *) second_client_session_handle,
     (ub4) OCI_HTYPE_SESSION, (void *) "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((void *)application_server_service_handle,
     (ub4) OCI_HTYPE_SVCCTX, (void *)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((void *)application_server_service_handle,
     (ub4) OCI_HTYPE_SVCCTX,
     (void *)second_client_session_handle, (ub4)0, (ub4)OCI_ATTR_SESSION,
     error_handle);
/*
** and then do operations as that client
*/
...
```

> **See Also:**
>
> - User Session Handle Attributes
> - OCIAttrSet()

## 10.5 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 can set default values and other session attributes through the `OCISessionBegin()` call, thus reducing server round-trips.

This section includes the following topics:

- Externally Initialized Context in OCI
- End-to-End Application Tracing
- Using OCISessionBegin() with an Externally Initialized Context

> **See Also:**
>
> - *Oracle Database Security Guide* the chapter about managing security for application developers
> - *Oracle Database SQL Language Reference* the `CREATE CONTEXT` statement and the `SYS_CONTEXT` function
> - OCIAttrSet()
> - OCISessionBegin()

### 10.5.1 Externally Initialized Context Attributes in OCI

The client applications you develop can set application contexts explicitly in the session handle before authentication.

To do this, use the following attributes in OCI functions:

- OCI_ATTR_APPCTX_SIZE
- OCI_ATTR_APPCTX_LIST
- Session Handle Attributes Used to Set an Externally Initialized Context

## 10.5.1.1 OCI_ATTR_APPCTX_SIZE

Use the `OCI_ATTR_APPCTX_SIZE` attribute to initialize the context array size with the desired number of context attributes in the `OCIAttrSet()` call.

This is shown in Example 10-9.

**Example 10-9 Defining the OCI_ATTR_APPCTX_SIZE Attribute to Initialize the Context Array Size with the Desired Number of Context Attributes**

```
OCIAttrSet(session, (ub4) OCI_HTYPE_SESSION,
           (void *)&size, (ub4)0, OCI_ATTR_APPCTX_SIZE, error_handle);
```

> **See Also:**
>
> OCIAttrSet()

## 10.5.1.2 OCI_ATTR_APPCTX_LIST

Use the `OCI_ATTR_APPCTX_LIST` attribute to get a handle on the application context list descriptor for the session in the `OCIAttrGet()` call.

This is shown in Example 10-10. (The parameter `ctxl_desc` must be of data type `OCIParam *`).

Example 10-11 shows how to use the application context list descriptor to obtain an individual descriptor for the i-th application context in a call to `OCIParamGet()`.

**Example 10-10 Using the OCI_ATTR_APPCTX_LIST Attribute to Get a Handle on the Application Context List Descriptor for the Session**

```
OCIAttrGet(session, (ub4) OCI_HTYPE_SESSION,
           (void *)&ctxl_desc, (ub4)0, OCI_ATTR_APPCTX_LIST, error_handle);
```

**Example 10-11 Calling OCIParamGet() to Obtain an Individual Descriptor for the i-th Application Context Using the Application Context List Descriptor**

```
OCIParamGet(ctxl_desc, OCI_DTYPE_PARAM, error_handle,(void **)&ctx_desc, i);
```

> **See Also:**
>
> - OCIAttrGet()
> - OCIParamGet()

### 10.5.1.3 Session Handle Attributes Used to Set an Externally Initialized Context

Shows how to set the appropriate session handle attributes values for the application 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. Example 10-12 shows the calls you can use to set them.

Note that only character type is supported, because application context operations are based on the `VARCHAR2` data type.

**Example 10-12 Defining Session Handle Attributes to Set Externally Initialized Context**

```
OCIAttrSet(ctx_desc, OCI_DTYPE_PARAM,
     (void *)ctx_name, sizeof(ctx_name), OCI_ATTR_APPCTX_NAME, error_handle);

OCIAttrSet(ctx_desc, OCI_DTYPE_PARAM,
     (void *)attr_name, sizeof(attr_name), OCI_ATTR_APPCTX_ATTR, error_handle);

OCIAttrSet(ctx_desc, OCI_DTYPE_PARAM,
     (void *)value, sizeof(value), OCI_ATTR_APPCTX_VALUE, error_handle);
```

**See Also:**

User Session Handle Attributes

## 10.5.2 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 by 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.

This section includes the following topics:

- OCI_ATTR_COLLECT_CALL_TIME
- OCI_ATTR_CALL_TIME
- Attributes for End-to-End Application Tracing

### 10.5.2.1 OCI_ATTR_COLLECT_CALL_TIME

Use the attribute `OCI_ATTR_COLLECT_CALL_TIME` and set the value to `TRUE` to enable the server to measure each call time.

Set a `boolean` variable to `TRUE` or `FALSE`. After you set the `OCI_ATTR_COLLECT_CALL_TIME` 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.

> **See Also:**
>
> OCIAttrSet()

### 10.5.2.2 OCI_ATTR_CALL_TIME

Use the attribute `OCI_ATTR_CALL_TIME` to get the elapsed time of the last server call.

The elapsed time, in microseconds, of the last server call is returned in a `ub8` variable by calling `OCIAttrGet()` with the `OCI_ATTR_CALL_TIME` attribute. Example 10-13 shows how to do this in a code fragment.

**Example 10-13 Using the OCI_ATTR_CALL_TIME Attribute to Get the Elapsed Time of the Last Server Call**

```
boolean enable_call_time;
ub8 call_time;
...
enable_call_time = TRUE;
OCIAttrSet(session, OCI_HTYPE_SESSION, (void *)&enable_call_time,
           (ub4)0, OCI_ATTR_COLLECT_CALL_TIME,
           (OCIError *)error_handle);
OCIStmtExecute(...);
OCIAttrGet(session, OCI_HTYPE_SESSION, (void *)&call_time,
           (ub4)0, OCI_ATTR_CALL_TIME,
           (OCIError *)error_handle);
...
```

> **See Also:**
>
> OCIAttrGet()

### 10.5.2.3 Attributes for End-to-End Application Tracing

Use these attributes for monitoring, tracing, and debugging applications.

Set these attributes for monitoring, 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_DBOP` - Name of the database operation set by the client application to be monitored in the database. Set to `NULL` if you want to end monitoring the current running database operation.
- `OCI_ATTR_CLIENT_INFO` - Client application additional information.

> **See Also:**
>
> User Session Handle Attributes
>
> Monitoring Database Operations in *Oracle Database SQL Tuning Guide*
>
> Monitoring Real-Time Database Performance in *Oracle Database 2 Day + Performance Tuning Guide*

## 10.5.3 Using OCISessionBegin() with an Externally Initialized Context

When you call `OCISessionBegin()`, the context set in the session handle is pushed to the server.

No additional contexts are propagated to the server session. Example 10-14 illustrates the use of these calls and attributes.

**Example 10-14 Using OCISessionBegin() with an Externally Initialized Context**

```
#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;
  void     *parmdp;
  ub4        ctxsize;
  OCIParam  *ctxldesc;
  OCIParam  *ctxedesc;

  OCIEnvCreate(&envhp, OCI_DEFAULT, (void *)0, 0, 0, 0,
                            (size_t)0, (void *)0);

  (void) OCIHandleAlloc( (void *) envhp, (void **) &errhp, OCI_HTYPE_ERROR,
                   (size_t) 0, (void **) 0);

  /* server contexts */
  (void) OCIHandleAlloc( (void *) envhp, (void **) &srvhp, OCI_HTYPE_SERVER,
                   (size_t) 0, (void **) 0);

  (void) OCIHandleAlloc( (void *) envhp, (void **) &svchp, OCI_HTYPE_SVCCTX,
                   (size_t) 0, (void **) 0);

  (void) OCIServerAttach( srvhp, errhp, (OraText *)"", strlen(""), 0);

  /* set attribute server context in the service context */
  (void) OCIAttrSet( (void *) svchp, OCI_HTYPE_SVCCTX, (void *)srvhp,
                    (ub4) 0, OCI_ATTR_SERVER, (OCIError *) errhp);

  (void) OCIHandleAlloc((void *) envhp, (void **)&authp,
                        (ub4) OCI_HTYPE_SESSION, (size_t) 0, (void **) 0);
/****************************************/
  /* set app ctx size to 2 because you want to set up 2 application contexts */
  ctxsize = 2;

  /* set up app ctx buffer */
  (void) OCIAttrSet((void *) authp, (ub4) OCI_HTYPE_SESSION,
                 (void *) &ctxsize, (ub4) 0,
                 (ub4) OCI_ATTR_APPCTX_SIZE, errhp);

  /* retrieve the list descriptor */
  (void) OCIAttrGet((void *)authp, (ub4) OCI_HTYPE_SESSION,
                    (void *)&ctxldesc, 0, OCI_ATTR_APPCTX_LIST, errhp );

  /* retrieve the 1st ctx element descriptor */
  (void) OCIParamGet(ctxldesc, OCI_DTYPE_PARAM, errhp, (void**)&ctxedesc, 1);

  (void) OCIAttrSet((void *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                 (void *) "HR", (ub4) strlen((char *)"HR"),
                 (ub4) OCI_ATTR_APPCTX_NAME, errhp);

  (void) OCIAttrSet((void *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                 (void *) "ATTR1", (ub4) strlen((char *)"ATTR1"),
                 (ub4) OCI_ATTR_APPCTX_ATTR, errhp);

  (void) OCIAttrSet((void *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                 (void *) "VALUE1", (ub4) strlen((char *)"VALUE1"),
                 (ub4) OCI_ATTR_APPCTX_VALUE, errhp);

  /* set second context */
  (void) OCIParamGet(ctxldesc, OCI_DTYPE_PARAM, errhp, (void**)&ctxedesc, 2);

  (void) OCIAttrSet((void *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                 (void *) "HR", (ub4) strlen((char *)"HR"),
                 (ub4) OCI_ATTR_APPCTX_NAME, errhp);

  (void) OCIAttrSet((void *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                 (void *) "ATTR2", (ub4) strlen((char *)"ATTR2"),
                 (ub4) OCI_ATTR_APPCTX_ATTR, errhp);

  (void) OCIAttrSet((void *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                 (void *) "VALUE2", (ub4) strlen((char *)"VALUE2"),
                 (ub4) OCI_ATTR_APPCTX_VALUE, errhp);
/*****************************************/
  (void) OCIAttrSet((void *) authp, (ub4) OCI_HTYPE_SESSION,
                 (void *) username, (ub4) strlen((char *)username),
                 (ub4) OCI_ATTR_USERNAME, errhp);

  (void) OCIAttrSet((void *) authp, (ub4) OCI_HTYPE_SESSION,
                 (void *) password, (ub4) strlen((char *)password),
                 (ub4) OCI_ATTR_PASSWORD, errhp);

  OCISessionBegin ( svchp,  errhp, authp, OCI_CRED_EXT, (ub4) OCI_DEFAULT);

}
```

> **See Also:**
>
> OCISessionBegin()

## 10.6 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, virtual private database (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;
```

To modify the data grouped in that namespace, users must 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 you want a human resources application to store an end-user's responsibility information in the user session, then it could create an `nm` namespace and an attribute called "responsibility" that can be assigned a value such as "manager" or "accountant". This is referred to as the *set operation* in this document.

If you want the application 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.

To clear all information in the `nm` namespace, the application can 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 can 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.

This section includes the following topics:

- Using Multiple SET Operations
- Using CLEAR-ALL Operations Between SET Operations
- Network Transport and PL/SQL on Client Namespace

## 10.6.1 Using Multiple SET Operations

Use the `OCIAppCtxSet()` function to perform a series of set operations on the "responsibility" attribute in the `CLIENTCONTEXT` namespace.

When this information is sent to the server, the latest value prevails 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 fragment shown in Example 10-15, on the client side. When this information is transported to the server, the server shows the latest value "vp" for the "responsibility" attribute in the `CLIENTCONTEXT` namespace.

You can clear specific attribute information in a client namespace. This can be done by setting the value of an attribute to `NULL` or to an empty string, as shown in Example 10-16 using the `OCIAppCtxSet()` function.

**Example 10-15 Changing the "responsibility" Attribute Value in the CLIENTCONTEXT Namespace**

```
err = OCIAppCtxSet((void *) sesshndl,(void *)"CLIENTCONTEXT",(ub4) 13,
                   (void *)"responsibility", 14
                   (void *)"manager", 7, errhp, OCI_DEFAULT);
err = OCIAppCtxSet((void *) sesshndl, (void*)"CLIENTCONTEXT", 13,
                   (void *)"responsibility", 14,(void *)"vp",2, errhp,
                   OCI_DEFAULT);
```

**Example 10-16 Two Ways to Clear Specific Attribute Information in a Client Namespace**

```
(void) OCIAppCtxSet((void *) sesshndl, (void *)"CLIENTCONTEXT", 13,
                    (void *)"responsibility", 14, (void *)0, 0,errhp,
                    OCI_DEFAULT);

(void) OCIAppCtxSet((void *) sesshndl, (void *)"CLIENTCONTEXT", 13
                    (void *)"responsibility", 14, (void *)"", 0,errhp,
                     OCI_DEFAULT);
```

**See Also:**

OCIAppCtxSet()

## 10.6.2 Using 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 several set operations, then values of all attributes in that namespace that were set before this clear-all operation are cleaned up on the client side and the server side. Only the set operations that were done after the clear-all operation are reflected on the server side. On the client side, the code appears, as shown in Example 10-17.

The clear-all operation clears any information set by earlier operations in the namespace `CLIENTCONTEXT`: "responsibility" = "manager" is removed. The information that was set subsequently will not be reflected on the server side.

**Example 10-17    Clearing All the Context Information in a Specific Client Namespace**

```
err = OCIAppCtxSet((void *) sesshndl,(void *)"CLIENTCONTEXT", 13,
                   (void *)"responsibility", 14,
                   (void *)"manager", 7,errhp, OCI_DEFAULT);
err = OCIAppCtxClearAll((void *) sesshndl, (void *)"CLIENTCONTEXT", 13, errhp,
                        OCI_DEFAULT);
err = OCIAppCtxSet((void *) sesshndl, (void*)"CLIENTCONTEXT",13
                   (void *)"office",6, (void *)"2op123", 5, errhp, OCI_DEFAULT);
```

**See Also:**

OCIAppCtxClearAll()

## 10.6.3 Network Transport and PL/SQL on Client Namespace

It is possible that an application could send application context information on an `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 applies 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 overwrites the information set provided by the fast network transport mechanism. If an error occurs in the network transport call, the main call is not executed.

However, an error in the main call does not affect the processing of the network transport call. Once the network transport call is processed, then there is no way to undo it. 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 call from an error in the main call. The client should not assume that an error from the main call will undo the round-trip network processing and should implement appropriate exception-handling mechanisms to prevent any inconsistencies.

**See Also:**

- OCIStmtExecute()
- OCIAppCtxClearAll()
- OCIAppCtxSet()

## 10.7 Using Edition-Based Redefinition

An edition provides a staging area where "editionable" objects changed by an application patch can be installed and executed while the existing application is still available.

You can specify an edition other than the database default by setting the attribute `OCI_ATTR_EDITION` at session initiation time. The application can call `OCIAttrSet()` specifying this attribute name and the edition as the value, as shown in Example 10-18.

If `OCIAttrSet()` is not called, the value of the edition name is obtained from the operating system environment variable `ORA_EDITION`. If that variable is not set, then the value of `OCI_ATTR_EDITION` is the empty string. If a nonempty value was specified, then the server sets the specified edition for the session, or the session uses the database default edition. The server then checks that the user has the `USE` privilege on the edition. If not, then the connect fails. If a nonexistent edition name was specified, then an error is returned.

**Example 10-18 Calling OCIAttrSet() to Set the OCI_ATTR_EDITION Attribute**

```
static void workerFunction()
{
  OCISvcCtx *svchp = (OCISvcCtx *) 0;
  OCIAuthInfo *authp = (OCIAuthInfo *)0;
  sword err;
  err =  OCIHandleAlloc((void *) envhp, (void **)&authp,
                        (ub4) OCI_HTYPE_AUTHINFO,
                        (size_t) 0, (void **) 0);
  if (err)
    checkerr(errhp, err);

  checkerr(errhp, OCIAttrSet((void *) authp, (ub4) OCI_HTYPE_AUTHINFO,
           (void *) username, (ub4) strlen((char *)username),
           (ub4) OCI_ATTR_USERNAME, errhp));

  checkerr(errhp,OCIAttrSet((void *) authp, (ub4) OCI_HTYPE_AUTHINFO,
           (void *) password, (ub4) strlen((char *)password),
           (ub4) OCI_ATTR_PASSWORD, errhp));

  (void) OCIAttrSet((void *) authp, (ub4) OCI_HTYPE_SESSION,
                  (void *) "Patch_Bug_12345",
                  (ub4) strlen((char *)"Patch_Bug_12345"),
                  (ub4) OCI_ATTR_EDITION, errhp);
  printf(("Create a new session that connects to the specified edition\n");
  if (err = OCISessionGet(envhp, errhp, &svchp, authp,
              (OraText *)connstr, (ub4)strlen((char *)connstr), NULL,
              0, NULL, NULL, NULL, OCI_DEFAULT))
  {
    checkerr(errhp, err);
    exit(1);
  }

  checkerr(errhp, OCISessionRelease(svchp, errhp, NULL, (ub4)0, OCI_DEFAULT));

  OCIHandleFree((void *)authp, OCI_HTYPE_AUTHINFO);
}
```

**See Also:**

- OCI_ATTR_EDITION in OCI_ATTR_EDITION
- *Oracle Database Development Guide* for a more complete description of edition-based redefinition
- Restrictions on Attributes Supported for OCI Session Pool in OCISessionGet()
- OCIAttrSet()

# 10.8 OCI Security Enhancements

The following security enhancements use configured parameters in the `init.ora` file or the `sqlnet.ora` file (the latter file is specifically noted for that feature).

These security enhancements are described in more detail in *Oracle Database Security Guide*. These initialization parameters apply to all instances of the database.

**See Also:**

*Oracle Database Security Guide*, the section about embedding calls in middle-tier applications to get, set, and clear client session IDs

This section includes the following topics:

- Controlling the Database Version Banner Displayed
- Banners for Unauthorized Access and User Actions Auditing
- Non-Deferred Linkage

## 10.8.1 Controlling the Database Version Banner Displayed

The `OCIServerVersion()` function can be issued before authentication (on a connected server handle after calling `OCIServerAttach()`) to get the database version.

To avoid disclosing the database version string before authentication, set the `SEC_RETURN_SERVER_RELEASE_BANNER` initialization parameter to `NO`. For example:

```
SEC_RETURN_SERVER_RELEASE_BANNER = NO
```

This displays the following string for Oracle Database Release 11.1 and all subsequent 11.1 releases and patch sets:

```
Oracle Database 11g Release 11.1.0.0.0 - Production
```

Set `SEC_RETURN_SERVER_RELEASE_BANNER` to `YES` and then the current banner is displayed. If you have installed Oracle Database Release 11.2.0.2, the banner displayed is:

```
Oracle Database 11g Enterprise Edition Release 11.2.0.2 - Production
```

This feature works with an Oracle Database Release 11.1 or later server, and any version client.

> **See Also:**
>
> - OCIServerVersion()
> - OCIServerAttach()

## 10.8.2 Banners for Unauthorized Access and User Actions Auditing

The following systemwide parameters are in `sqlnet.ora` and warn users against unauthorized access and possible auditing of user actions.

These features are available in Oracle Database Release 11.1 and later servers and clients. The content of the banners is in text files that the database administrator creates. There is a 512 byte buffer limit for displaying the banner content. If this buffer limit is exceeded, the banner content will appear to be cut off. The access banner syntax is:

```
SEC_USER_UNAUTHORIZED_ACCESS_BANNER = file_path1
```

In this syntax, *`file_path1`* is the path of a text file. To retrieve the banner, get the value of the attribute `OCI_ATTR_ACCESS_BANNER` from the server handle after calls to either `OCIServerAttach()` or `OCISessionGet()`.

The audit banner syntax is:

```
SEC_USER_AUDIT_ACTION_BANNER = file_path2
```

In this syntax, *`file_path2`* is the path of a text file. To retrieve the banner, get the value of the attribute `OCI_ATTR_AUDIT_BANNER` from the session handle after calls to either `OCISessionBegin()`, `OCISessionGet()`, `OCILogon()`, or `OCILogon2()`.

**See Also:**

- OCI_ATTR_ACCESS_BANNER
- OCI_ATTR_AUDIT_BANNER
- OCIServerAttach()
- OCISessionGet()
- OCISessionBegin()
- OCILogon()
- OCILogon2()

## 10.8.3 Non-Deferred Linkage

Non-deferred linkage of applications is no longer supported and the `Makefile` is modified to remove it.

This method of linking was used before OCI V7.

# 10.9 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, system registers, and stacks. Global and static variables are common to all threads, and a mutual exclusion 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 exclusion locks). This mechanism 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 Oracle Database and the OCI libraries allows developers to use OCI in a multithreaded environment. Thread safety ensures that 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 Database system-specific documentation for more information.

In a multithreaded Linux or UNIX environment, OCI calls except `OCIBreak()` are not allowed in a user signal handler.

The correct way to use and free handles is to create the handle, use the handle, then free the handle only after all the threads have been destroyed, when the application is terminating.

This section includes the following topics:

- Advantages of OCI Thread Safety
- OCI Thread Safety and Three-Tier Architectures
- About Implementing Thread Safety

**See Also:**

OCIBreak()

## 10.9.1 Advantages of OCI Thread Safety

Lists the advantages of implementing 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.

## 10.9.2 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 (client-agent-server) architectures.

In this architecture, the client is concerned only with presentation services. The agent (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 application server (agent) is very well suited to being a multithreaded application server, with each thread serving a single client application. In an Oracle Database environment, this application server is an OCI or precompiler program.

## 10.9.3 About Implementing Thread Safety

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 `OCIEnvCreate()` 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 `OCIEnvCreate()` 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 manages 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_ENV_NO_MUTEX` in the `mode` parameter of either the `OCIEnvCreate()` or `OCIEnvNlsCreate()` calls.

The following scenarios are possible, depending on how many connections exist in each environment handle, and how many threads are spawned in each connection.

- If an application has multiple environment handles, with a single thread in each, mutexes are not required.
- 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_ENV_NO_MUTEX` for the `mode` of `OCIEnvNlsCreate()`. The application must set mutual exclusion locks (mutex) for 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 ensure 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:**
>
> Most processing of an OCI call happens on the server, so if two threads using OCI calls go to the same connection, then one of them can be blocked while the other finishes processing at the server.
>
> Use one error handle for each thread in an application, because OCI errors can be overwritten by other threads.

This section includes the following topics:

- About Polling Mode Operations and Thread Safety
- Mixing 7.x and Later Release OCI Calls

> **See Also:**
>
> - OCIEnvNlsCreate()
> - OCIEnvCreate()

### 10.9.3.1 About Polling Mode Operations and Thread Safety

OCI supports polling mode operations.

When OCI is operating in threaded mode, OCI calls that poll for completion acquire mutexes when the OCI call is actively executing. However, when OCI returns control to the application, OCI releases any acquired mutexes. The caller should ensure that no other OCI call is made on the connection until the polling mode OCI operation in progress completes.

> **See Also:**
>
> Polling Mode Operations in OCI

### 10.9.3.2 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 gets 7.x behavior on any subsequent 7.x function calls.

## 10.10 OCIThread Package

The `OCIThread` package provides some commonly used threading primitives.

The `OCIThread` package 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 are only able to support a limited implementation of the `OCIThread` package. As a result, products that rely on all of the `OCIThread` functionality do not port to all operating systems. Products that must be ported to all operating systems must use only a subset of the `OCIThread` 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()` 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 although 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 do not require a multiple-thread environment to work correctly. They are 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 must determine at run time whether you are in a multithreaded environment, call `OCIThreadIsMulti()` before using an `OCIThread` active threading primitive.

  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 run time with the `OCIThreadIsMuilt()` call.

**See Also:**

- Thread Management Functions
- `cdemothr.c` in the `demo` directory is an example of a multithreading application
- OCIThreadProcessInit()
- OCIThreadInit()
- OCIThreadTerm()
- OCIThreadIsMulti()

## 10.10.1 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 you can use any of its functionality.

The observed behavior of the initialization and termination functions is the same regardless of whether `OCIThread` is in a single-threaded or a multithreaded environment. Table 10-6 lists functions for thread initialization and termination.

**Table 10-6 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 |

This section includes the following topic: OCIThread Context.

**See Also:**

Thread Management Functions

### 10.10.1.1 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.

**See Also:**

- OCIThreadInit()
- OCIThreadTerm()

## 10.10.2 Passive Threading Primitives

Lists the passive threading primitive functions.

The passive threading primitives deal with the manipulation of mutex, thread IDs, and thread-specific data. Because 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 10-7 lists functions used to implement passive threading.

**Table 10-7 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 generates a new 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 one 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` |

This section includes the following topics:

- OCIThreadMutex
- OCIThreadKey

- OCIThreadKeyDestFunc
- OCIThreadId

## 10.10.2.1 OCIThreadMutex

The `OCIThreadMutex` data type is used to represent a mutex.

This mutex is used to ensure that either:

- Only one thread accesses a given set of data at a time
- 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()`.

**See Also:**

- OCIThreadMutexInit()
- OCIThreadMutexDestroy()
- OCIThreadMutexAcquire()
- OCIThreadMutexRelease()

## 10.10.2.2 OCIThreadKey

The data type `OCIThreadKey` can be thought of as a process-wide variable with a thread-specific value.

Thus 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 is always the same as the value that it last set for the key. It does not see any values set for the key by other threads. The data type of the value held by a key is a `void *` 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 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 to the threads pool.

Later, the threads pool manager assigns another thread to perform some operations on the task, and the task must retrieve the data it stored earlier in initialization. Because the task is running in another thread, it is not able to retrieve the same data. Application developers that use thread pools must be aware of this.

> **See Also:**
>
> - OCIThreadKeyInit()
> - OCIThreadKeySet()
> - OCIThreadKeyGet()

## 10.10.2.3 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 is 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.

> **See Also:**
>
> OCIThreadKeyInit()

## 10.10.2.4 OCIThreadId

`OCIThreadId` data type is used to identify a thread.

At any given time, no two threads can have the same `OCIThreadId`, but `OCIThreadId` values can be recycled; after 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 can never be the same as the ID of an actual thread.

**See Also:**

OCIThreadIdGet()

## 10.10.3 Active Threading Primitives

Lists the active threading primitives, which 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 10-8 lists functions used to implement active threading.

**Table 10-8 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 |

This section includes the following topic: OCIThreadHandle.

**See Also:**

OCIThreadIsMulti()

### 10.10.3.1 OCIThreadHandle

Data type `OCIThreadHandle` is used to manipulate a thread in the active primitives, `OCIThreadJoin()` and `OCIThreadClose()`.

A thread handle opened by `OCIThreadCreate()` must be closed in a matching call to `OCIThreadClose()`. A thread handle is invalid after the call to `OCIThreadClose()`.

**See Also:**

- OCIThreadJoin()
- OCIThreadClose()
- OCIThreadCreate()

11

# Session Pooling and Connection Pooling in OCI

This chapter describes OCI session pooling and connection pooling features.

This chapter contains these topics:

- Session Pooling in OCI
- Database Resident Connection Pooling
- About Using Oracle Connection Manager in Traffic Director Mode
- Connection Pooling in OCI
- When to Use Connection Pooling, Session Pooling, or Neither

## 11.1 Session Pooling in OCI

*Session pooling* means that the application creates and maintains a group of stateless sessions to the database.

These sessions are provided 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 releases 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 a default session, set certain attributes on it, label it or tag it, and return it to the pool. That user, or some other user, can require a session with the same attributes, and thus request 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.

Proxy sessions, too, can be created and maintained through session pooling in OCI.

The behavior of the application when no free sessions are available and the pool has reached its maximum size depends 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 session pooling is 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.

This section includes the following topics:

- Functionality of OCI Session Pooling
- Homogeneous and Heterogeneous Session Pools
- About Using Tags in Session Pools
- OCI Handles for Session Pooling

- Using OCI Session Pooling
- OCI Calls for Session Pooling
- Example of OCI Session Pooling

> **See Also:**
>
> About Using Tags in Session Pools

## 11.1.1 Functionality of OCI Session Pooling

Describes tasks that session pooling can perform.

Session pooling can perform the following tasks:

- 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 a very long time, and creating sessions when required.
- Allow for session pooling with authentication.

## 11.1.2 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 for authentication (they have the same user name, password, and privileges). *Heterogeneous* session pooling means that you must provide authentication information because the sessions can have different security attributes and privileges.

## 11.1.3 About Using Tags in Session Pools

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 NLS settings), and return the session to the pool, labeling it with an appropriate tag in the `OCISessionRelease()` call.

The original user, or some other user, can request a session with the same tags to have a session with the same attributes, and can do so by providing the same tag in the `OCISessionGet()` call.

This section includes the following topic: Multi-Property Tags.

**See Also:**

- OCISessionRelease()
- OCISessionGet()

### 11.1.3.1 Multi-Property Tags

Beginning with 12*c* Release 2 (12.2), a tag can have multiple properties. This is referred to as a multi-property tag.

A multi-property tag is comprised of one or more <property-name>=<property-value> pairs separated by a semi-colon, where <property-name>=<property-value> are both strings.

During an `OCISessionGet()` call, in the `taginfo` parameter, the property name appearing first is given the highest property for finding a match and the property name appearing last is given the lowest priority. Therefore the ordering of the properties in the string is significant in determining a matching session in the pool. The example that follows below the list of restrictions illustrates this point.

This functionality also works with DRCP.

The following restrictions apply to property names and property values that can be passed in a multi-property tag:

- Both the property name and the property value are case sensitive.
- A property name can occur only once in a tag. In case the same property name is specified more than once, an error will be thrown.
- A non empty string should be specified for both the property name and the value.
- Leading and trailing spaces before and after a property name and leading and trailing space before and after a property value will be truncated. For example, “PDB = PDB1” is treated as “PDB=PDB1”.
- There should be no white spaces in the property name and the property value. For example, NLS <space> LANGUAGE=French will result in an error because of the space between NLS and LANGUAGE.

To explain the notion of multiple properties, assume that the application to be deployed in a CDB environment requires that the session get requests should be satisfied with sessions from the same pluggable database (for example, pdb1 ) as much as possible. Next, it also requires that the sessions belong to the same language (for example, FRENCH), but gives a higher priority to sessions to pdb1. The application can then provide a multi-property-tag as follows:

```
char *props  = “PDB=pdb1;LANGUAGE=FRENCH”
```

Now, assume that there are two sessions in the pool with properties as shown:

```
Session 1 = >  “PDB=pdb1;LANGUAGE=CHINESE”


Session 2 = >  “PDB=pdb2;LANGUAGE=FRENCH”
```

In this situation, the session get request (`OCISessionGet()`) returns Session 1 because the `PDB` property implicitly carries a higher priority by being placed ahead of the `LANGUAGE` property.

> **See Also:**
>
> OCISessionGet() for a further discussion of tagging sessions

This section includes the following topic: PL/SQL Callback for Session State Fix Up.

### 11.1.3.1.1 PL/SQL Callback for Session State Fix Up

When using multi-property tags, a PL/SQL based fix-up callback for the session state can be provided on the server.

This application-provided callback transforms a session checked out from the pool to the desired state requested by the application as indicated by the multi-property tag. This callback works with or without Database Resident Connection Pooling (DRCP).

Using this callback can improve the performance of your application because the fix-up logic is run for the session state on the server. So, this feature eliminates application round-trips to the database for the fix-up logic. The callback is supplied by the user who connects using `OCISessionGet()`. The callback must be provided as an attribute `OCI_ATTR_FIXUP_CALLBACK` on the authentication handle passed to `OCISessionGet()` for applications not using OCISessionPool or using custom pools. For applications using OCISessionPool this attribute must be set on the authentication handle, which in turn must be set on the session pool handle as the attribute `OCI_ATTR_SPOOL_AUTH`.

**Example 11-1    Example of PL/SQL Fix-Up Callback**

The following PL/SQL fix-up callback example code snippet handles tags whose `key=value` properties can be used directly in an ALTER SESSION statement, such as `TIME_ZONE=UTC;NLS_DATE_FORMAT=DD-MM-YYYY`:

```
CREATE OR REPLACE PACKAGE myPackage AS
  TYPE property_t IS TABLE OF VARCHAR2(64) INDEX BY VARCHAR2(64);
  PROCEDURE buildTab(
    tag          IN  VARCHAR2,
    propertyTab  OUT property_t
  );
  PROCEDURE myPlsqlCallback (
    requestedTag IN  VARCHAR2,
    actualTag    IN  VARCHAR2
  );
END;
```

```
/

CREATE OR REPLACE PACKAGE BODY myPackage AS

  -- Parse the "property=value" pairs in the tag
  PROCEDURE buildTab(tag IN VARCHAR2, propertyTab OUT property_t) IS
    property  VARCHAR2(64);
    propertyName  VARCHAR2(64);
    propertyValue VARCHAR2(64);
    propertyEndPos NUMBER := 1;
    propertyStartPos NUMBER := 1;
    propertyNameEndPos NUMBER := 1;
  begin
    WHILE (LENGTH(tag) > propertyEndPos)
    LOOP
      propertyEndPos := INSTR(tag, ';', propertyStartPos);
      IF (propertyEndPos = 0) THEN
        propertyEndPos := LENGTH(tag) + 1;
      END IF;
      propertyNameEndPos := INSTR(tag, '=', propertyStartPos);
      propertyName := SUBSTR(tag, propertyStartPos,
                   propertyNameEndPos - propertyStartPos);
      propertyValue := SUBSTR(tag, propertyNameEndPos + 1,
                    propertyEndPos - propertyNameEndPos - 1);
      propertyTab(propertyName) := propertyValue;
      propertyStartPos := propertyEndPos + 1;
    END LOOP;
  END;

  PROCEDURE myPlsqlCallback (
    requestedTag IN VARCHAR2,
    actualTag IN VARCHAR2
  ) IS
    reqPropTab property_t;
    actPropTab property_t;
    propertyName VARCHAR2(64);
  BEGIN
    buildTab(requestedTag, reqPropTab);
    buildTab(actualTag, actPropTab);

    -- Iterate over requested properties to set state when it's not
    -- currently set, or not set to the desired value
    propertyName := reqPropTab.FIRST;
    WHILE (propertyName IS NOT NULL)
    LOOP
      IF ((NOT actPropTab.exists(propertyName)) OR
         (actPropTab(propertyName) != reqPropTab(propertyName))) THEN
        EXECUTE IMMEDIATE 'ALTER SESSION SET ' || propertyName || '=''' ||
reqPropTab(propertyName) || '''';
      END IF;
      propertyName := reqPropTab.NEXT(propertyName);
    END LOOP;
    -- Could iterate over other actual props to reset any extra props to a
default state
  END;
```

```
END myPackage;
/
```

**See Also:**

- Authentication Information Handle Attributes for more information about `OCI_ATTR_FIXUP_CALLBACK`
- Session Pool Handle Attributes for more information about `OCI_ATTR_SPOOL_AUTH`

## 11.1.4 OCI Handles for Session Pooling

What are the handle types for session pooling.

The following handle types are for session pooling:

- OCISPool
- OCIAuthInfo

### 11.1.4.1 OCISPool

This is the session pool handle.

It is allocated using `OCIHandleAlloc()`. It must be passed to `OCISessionPoolCreate()` and `OCISessionPoolDestroy()`. It has the attribute type `OCI_HTYPE_SPOOL`.

An example of the `OCIHandleAlloc()` call follows:

```
OCISPool *spoolhp;
OCIHandleAlloc((void *) envhp, (void **) &spoolhp, OCI_HTYPE_SPOOL,
                        (size_t) 0, (void **) 0));
```

For an environment handle, multiple session pools can be created.

**See Also:**

- OCIHandleAlloc()
- OCISessionPoolCreate()
- OCISessionPoolDestroy()

### 11.1.4.2 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 a user session handle. The authentication information handle has the attribute type `OCI_HTYPE_AUTHINFO` (see Table 4-1).

An example of the `OCIHandleAlloc()` call follows:

```
OCIAuthInfo *authp;
OCIHandleAlloc((void *) envhp, (void **) &authp, OCI_HTYPE_AUTHINFO,
                         (size_t) 0, (void **) 0));
```

**See Also:**

- OCIHandleAlloc()
- User Session Handle Attributes for the attributes that belong to the authentication information handle
- Session Pool Handle Attributes for more information about the session pooling attributes
- Connect, Authorize, and Initialize Functions for complete information about the functions used in session pooling
- See "OCISessionGet()" for details of the session handle attributes that you can use with this call

## 11.1.5 Using OCI Session Pooling

Shows the steps to write a simple session pooling application that uses a user name and password.

The steps in writing a simple session pooling application that uses a user name and password are as follows:

1. Allocate the session pool handle using `OCIHandleAlloc()` for an `OCISPool` handle. Multiple session pools can be created for an environment handle.
2. Create the session pool using `OCISessionPoolCreate()` with `mode` set to `OCI_DEFAULT` (for a new session pool). See the function for a discussion of the other modes.
3. Loop for each thread. Create the thread with a function that does the following:
    a. Allocates an authentication information handle of type `OCIAuthInfo` using `OCIHandleAlloc()`
    b. Sets the user name and password in the authentication information handle using `OCIAttrSet()`
    c. Gets a pooled session using `OCISessionGet()` with `mode` set to `OCI_SESSGET_SPOOL`
    d. Performs the transaction
    e. Allocates the handle
    f. Prepares the statement

> **Note:**
>
> When using service contexts obtained from OCI session pool, you are required to use the service context returned by `OCISessionGet()` (or `OCILogon2()`), and not create other service contexts outside of these calls.
>
> Any statement handle obtained using `OCIStmtPrepare2()` with the service context should be subsequently used only in conjunction with the same service context, and never with a different service context.

   g. Executes the statement
   h. Commits or rolls back the transactions
   i. Releases the session (log off) with `OCISessionRelease()`
   j. Frees the authentication information handle with `OCIHandleFree()`
   k. Ends the loop for each thread

4. Destroy the session pool using `OCISessionPoolDestroy()`.

> **See Also:**
>
> - OCISessionPoolCreate()
> - OCIHandleAlloc()
> - OCIAttrSet()
> - OCISessionGet()
> - OCILogon2()
> - OCIStmtPrepare2()
> - OCISessionRelease()
> - OCIHandleFree()
> - OCISessionPoolDestroy()

## 11.1.6 OCI Calls for Session Pooling

Describes the usages for OCI calls for session pooling.

OCI provides calls for session pooling to perform the following tasks:

- Allocate the Pool Handle
- Create the Connection Pool
- Log On to the Database
- Log Off from the Database
- Destroy the Connection Pool
- Free the Pool Handle

### 11.1.6.1 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((void *) envhp, (void **) &poolhp, OCI_HTYPE_SPOOL, (size_t) 0,
               (void **) 0));
```

> **See Also:**
>
> OCIHandleAlloc()

### 11.1.6.2 Create the Pool Session

You can use the function `OCISessionPoolCreate()` 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);
```

> **See Also:**
>
> OCISessionPoolCreate()

### 11.1.6.3 Log On to the Database

You can use these calls to log on to the database in session pooling mode.

- `OCILogon2()`

  This is the simplest call. However, it does not give the user the option of using tagging. Here is an example of how to use `OCILogon2()` 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 call to use. It gives the user the option of using tagging to label sessions in the pool, which makes 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);
```

When using service contexts obtained from an OCI session pool, you are required to use the service context returned by `OCISessionGet()` (or `OCILogon2()`), and not create other service contexts outside of these calls.

Any statement handle obtained using `OCIStmtPrepare2()` with the service context should be subsequently used only in conjunction with the same service context, and never with a different service context.

**See Also:**

- OCILogon2()
- OCISessionGet()
- OCIStmtPrepare2()

### 11.1.6.4 Log Off from the Database

Indicates two ways in which to log off from the database in session pooling mode depending on the logon call.

From the following calls, choose the one that corresponds to the logon call and use it to log off from the database in session pooling mode.

- `OCILogoff()`

  If you used `OCILogon2()` to make the connection, you must call `OCILogoff()` to log off.

- `OCISessionRelease()`

  If you used `OCISessionGet()` to make the connection, then you must call `OCISessionRelease()` to log off. Pending transactions are automatically committed.

**See Also:**

- OCILogoff()
- OCILogon2()
- OCISessionRelease()
- OCISessionGet()

### 11.1.6.5 Destroy the Session Pool

Call `OCISessionPoolDestroy()` to destroy the session pool.

This is shown in the following example:

```
OCISessionPoolDestroy(poolhp, errhp, OCI_DEFAULT);
```

> **See Also:**
>
> OCISessionPoolDestroy()

### 11.1.6.6 Free the Pool Handle

Call `OCIHandleFree()` to free the session pool handle.

This is shown in the following example:

```
OCIHandleFree((void *)poolhp, OCI_HTYPE_SPOOL);
```

> **Note:**
>
> Developers: You are advised to commit or roll back any open transaction before releasing the connection back to the pool. If this is not done, Oracle Database automatically commits any open transaction when the connection is released.
>
> If an instance failure is detected while the session pool is being used, OCI tries to clean up the sessions to that instance.

> **See Also:**
>
> OCIHandleFree()

### 11.1.7 Example of OCI Session Pooling

Where to find an example of session pooling in a tested complete program.

For an example of session pooling in a tested complete program, see `cdemosp.c` in directory `demo`.

## 11.2 Database Resident Connection Pooling

Database resident connection pooling (DRCP) provides a connection pool in the database server for typical web application usage scenarios where the application acquires a database connection, works on it for a relatively short duration, and then releases it.

DRCP pools server processes, each of which is the equivalent of a dedicated server process and a database session combined. (Henceforth these "dedicated" server processes are referred to as *pooled servers*.)

DRCP complements middle-tier connection pools that share connections between threads in a middle-tier process. In addition, DRCP enables sharing of database connections across middle-tier processes on the same middle-tier host and even across middle-tier hosts. This results in significant reduction in key database resources needed to support a large number of client connections, thereby reducing the database tier memory footprint and boosting the scalability of both middle-tier and database tiers. Having a pool of readily available servers has the additional benefit of reducing the cost of creating and tearing down client connections.

DRCP is especially relevant for architectures with multiprocess single-threaded application servers (such as PHP/Apache) that cannot do middle-tier connection pooling. Using DRCP, the database can scale to tens of thousands of simultaneous connections.

> **See Also:**
>
> *Oracle Database Development Guide* for complete information about DRCP

# 11.3 About Using Oracle Connection Manager in Traffic Director Mode

Oracle Connection Manager in Traffic Director Mode is a proxy that is placed between supported database clients and database instances.

Supported clients from Oracle Database 11*g* Release 2 (11.2) and later can connect to Oracle Connection Manager in Traffic Director Mode. Oracle Connection Manager in Traffic Director Mode provides improved high availability (HA) for planned and unplanned database server outages, connection multiplexing support, and load balancing. Support for Oracle Connection Manager in Traffic Director Mode is described in more detail in the following sections

- Modes of Operation
- Key Features

**Modes of Operation**

Oracle Connection Manager in Traffic Director Mode supports the following modes of operation:

- In pooled connection mode, Oracle Connection Manager in Traffic Director Mode supports any application using the following database client releases:
  - OCI, OCCI, and Open Source Drivers (Oracle Database 11*g* release 2 (11.2.0.4) and later))
  - JDBC (Oracle Database 12*c* release 1 (12.1) and later)
  - ODP.NET (Oracle Database 12*c* release 2 (12.2) and later)

  In addition, applications must use DRCP. That is, the application must enable DRCP in the connect string (or in the `tnsnames.ora` alias).

- In non-pooled connection (or dedicated) mode, Oracle Connection Manager in Traffic Director Mode supports any application using database client releases Oracle Database 11*g* release 2 (11.2.0.4) and later. In this mode, some capabilities, such as connection multiplexing are not available.

**Key Features**

Oracle Connection Manager in Traffic Director Mode furnishes support for the following:

- Transparent performance enhancements and connection multiplexing, which includes:
  - Statement caching, rows prefetching, and result set caching are auto-enabled for all modes of operation.
  - Database session multiplexing (pooled mode only) using the proxy resident connection pool (PRCP), where PRCP is a proxy mode of Database Resident Connection Pooling (DRCP). Applications get transparent connect-time load balancing and run-time load balancing between Oracle Connection Manager in Traffic Director Mode and the database.
  - For multiple Oracle Connection Manager in Traffic Director Mode instances, applications get increased scalability through client-side connect time load balancing or with a load balancer (BIG-IP, NGINX, and others)
- Zero application downtime
  - Planned database maintenance or pluggable database (PDB) relocation
    - Pooled mode

      Oracle Connection Manager in Traffic Director Mode responds to Oracle Notification Service (ONS) events for planned outages and redirects work. Connections are drained from the pool on Oracle Connection Manager in Traffic Director Mode when the request completes. Service relocation is supported for Oracle Database 11*g* release 2 (11.2.0.4) and later.

      For PDB relocation, Oracle Connection Manager in Traffic Director Mode responds to in-band notifications when a PDB is relocated, that is even when ONS is not configured (for Oracle Database release 18c, version 18.1 and later server only)
    - Non-pooled or dedicated mode

      When there is no request boundary information from the client, Oracle Connection Manager in Traffic Director Mode supports planned outage for many applications (as long as only simple session state and cursor state need to be preserved across the request/transaction boundaries). This support includes:
      - Stop service/PDB at the transaction boundary or it leverages Oracle Database release 18c continuous application availability to stop the service at the request boundary
      - Oracle Connection Manager in Traffic Director Mode leverages Transparent Application Failover (TAF) failover restore to reconnect and restore simple states.
  - Unplanned database outages for read-mostly workloads

- High Availability of Oracle Connection Manager in Traffic Director Mode to avoid a single point of failure. This is supported by:
  - Multiple instances of Oracle Connection Manager in Traffic Director Mode using a load balancer or client side load balancing/failover in the connect string
  - Rolling upgrade of Oracle Connection Manager in Traffic Director Mode instances
  - Graceful close of existing connections from client to Oracle Connection Manager in Traffic Director Mode for planned outages
  - In-band notifications to Oracle Database release 18c and later clients
  - For older clients, notifications are sent with the response of the current request
- For security and isolation, Oracle Connection Manager in Traffic Director Mode furnishes:
  - Database Proxy supporting transmission control protocol/transmission control protocol secure (TCP/TCPS) and protocol conversion
  - Firewall based on the IP address, service name, and secure socket layer/transport layer security (SSL/TLS) wallets
  - Tenant isolation in a multi-tenant environment
  - Protection against denial-of-service and fuzzing attacks
  - Secure tunneling of database traffic across Oracle Database on-premises and Oracle Cloud

**See Also:**

- *Oracle Database Net Services Administrator's Guide* for information about configuring `cman.ora` configuration file to set up Oracle Connection Manager in Traffic Director Mode
- *Oracle Database Net Services Administrator's Guide* for information about configuring databases for Oracle Connection Manager in Traffic Director Mode proxy authentication
- *Oracle Database Net Services Administrator's Guide* for information about configuring Oracle Connection Manager in Traffic Director Mode for unplanned down events
- *Oracle Database Net Services Administrator's Guide* for information about configuring Oracle Connection Manager in Traffic Director Mode for planned down events
- *Oracle Database Net Services Administrator's Guide* for information about configuring proxy resident connection pools for use by Oracle Connection Manager in Traffic Director Mode
- *Oracle Database Net Services Administrator's Guide* for information about functionality not supported for all drivers with Oracle Connection Manager in Traffic Director Mode
- *Oracle Database Net Services Reference* for an overview of Oracle CMAN configuration file

# 11.4 Connection Pooling in OCI

Connection pooling is the use of a group (the pool) of reusable physical connections by several sessions to balance loads.

The pool is managed by OCI, not the application. Applications that can use connection pooling include middle-tier applications for web application servers and email servers.

One use 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.

This section includes the following topics:

- OCI Connection Pooling Concepts
- Using OCI Calls for Connection Pooling
- Examples of OCI Connection Pooling

## 11.4.1 OCI Connection Pooling Concepts

Oracle Database has several transaction monitoring capabilities such as the fine-grained management of database sessions and connections. Fine-grained management of database sessions is done by separating the notion of database sessions (user handles) from connections (server handles). By using OCI calls for session switching and session migration, an application server or transaction monitor can multiplex several sessions over fewer physical connections, thus achieving a high degree of scalability by pooling 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.

This section includes the following topics:

- Similarities and Differences from a Shared Server
- Stateless Sessions Versus Stateful Sessions
- Multiple Connection Pools
- Transparent Application Failover

### 11.4.1.1 Similarities and Differences from a Shared Server

Connection pooling on the middletier is similar to what a 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 connection pool on the middle tier controls the pooling of dedicated server processes including incoming connections into the dedicated server processes. 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. However, the physical connection from the connection pool is established directly from the middletier to the dedicated server process in the back-end server pool.

Connection pooling is beneficial only if the middle tier 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.

### 11.4.1.2 Stateless Sessions Versus Stateful 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 a completely different request on behalf of a completely different three-tier user.

Stateful sessions to the database, however, 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 include open transactions, the fetch state from a statement, or a PL/SQL package state. So long as the state exists, the session is not reusable for a different request.

> **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) that allows the session to be serially reused for a different three-tier user (when such state is cleaned up).
>
> Stateless sessions are typically used in conjunction with statement caching.

What connection pooling offers is stateless connections and stateful sessions.

> **See Also:**
>
> Session Pooling in OCI if you must work with stateless sessions

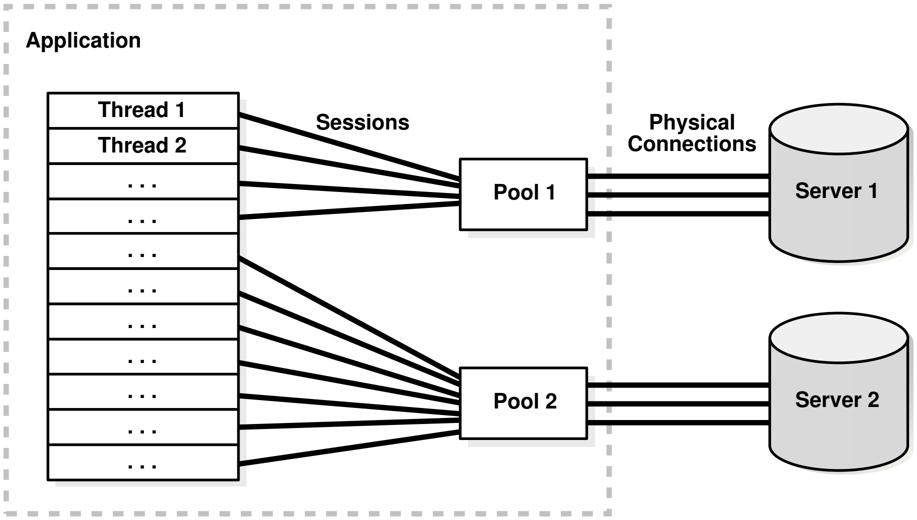
### 11.4.1.3 Multiple Connection Pools

You can use the advanced concept of multiple connection pools 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.

Figure 11-1 illustrates OCI connection pooling.

**Figure 11-1    OCI Connection Pooling**



### 11.4.1.4 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 fail over when they use the pool for a database round-trip after their instance failure. The listener is configured to route the connection to a good instance if available, as is typical with service-based connect strings.

> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide*, the chapter about configuring transparent application failover

## 11.4.2 Using OCI Calls for Connection Pooling

Lists the steps you must follow to use connection pooling in your application.

To use connection pooling in your application, you must:

1. Allocate the Pool Handle

2. Create the Connection Pool
3. Log On to the Database
4. Deal with SGA Limitations in Connection Pooling
5. Log Off from the Database
6. Destroy the Connection Pool
7. Free the Pool Handle

### 11.4.2.1 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((void *) envhp, (void **) &poolhp, OCI_HTYPE_CPOOL,
                      (size_t) 0, (void **) 0));
```

**See Also:**

OCIHandleAlloc()

### 11.4.2.2 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 waits until it gets one. However, if the `OCI_ATTR_CONN_NOWAIT` attribute is set for the pool, an error is returned.
- A `poolUsername` and a `poolPassword`, 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 active.

Because all the preceding attributes can be configured dynamically, the application can read the current load (number of open connections and number of busy connections) and tune 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` 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 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) connMin, (ub4) connMax, (ub4) connIncr,
                   (text *)poolUsername,strlen(poolUserLen),
                   (text *)poolPassword,strlen(poolPassLen),
                   OCI_DEFAULT));
```

**See Also:**

- OCIConnectionPoolCreate()
- OCI_ATTR_CONN_TIMEOUT
- OCIServerAttach()
- OCILogon2()

### 11.4.2.3 Log On to the Database

The application can use one of several interfaces.

The application must 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));

}
```

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()`

  You can use another interface if the application must set any special attributes on the user session handle and server handle. For such a requirement, applications must allocate all the handles (connection pool handle, server handles, session handles, and service context handles). You would follow this sequence:

  1. Create the connection pool.

     Connection pooling does the multiplexing of a virtual server handle over physical connections transparently, eliminating the need for users to do so. The user gets the feeling of a session having a dedicated (virtual) connection. Because the multiplexing is done transparently to the user, users must 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 for connection pooling and should not be used.

  2. Call `OCIServerAttach()` with mode set to `OCI_CPOOL`.

     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.

  3. Call `OCISessionBegin()` with mode set to `OCI_DEFAULT`.

     Credentials can be set to `OCI_CRED_RDBMS`, `OCI_CRED_EXT`, or `OCI_CRED_PROXY` using `OCISessionBegin()`. If the credentials are set to `OCI_CRED_EXT`, no user name and no password need to be set on the session handle. If the credentials are set to `OCI_CRED_PROXY`, only the user name must be set on the session handle. (no explicit primary session must be created and `OCI_ATTR_MIGSESSION` need not be set).

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 the `OCI_MIGRATE` flag 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 that are transparently multiplexed onto real connections.

**See Also:**

- OCILogon2()
- OCISessionGet()
- OCIServerAttach()
- OCISessionBegin()

### 11.4.2.4 Deal with SGA Limitations in Connection Pooling

With `OCI_CPOOL` mode (connection pooling), the session memory (UGA) in the back-end database comes 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 are similar to configuring the LARGE POOL in a shared server back end except that the instance is still in dedicated mode.

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 is not possible because sessions using dedicated database links are 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:**

- *Oracle Database Performance Tuning Guide*, the section about configuring a shared server
- *Oracle Database Administrator’s Guide*, the section on shared database links for more information about distributed databases

### 11.4.2.5 Log Off from the Database

Choose the appropriate call to log off from the database in connection pooling mode.

From the following calls, choose the one that corresponds to the logon call and use it 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 the session, then `OCISessionEnd()` must be called to end the session and `OCIServerDetach()` must be called to release the connection.

**See Also:**

- OCILogoff()
- OCILogon2()
- OCISessionRelease()
- OCISessionEnd()
- OCIServerDetach()
- OCIServerAttach()
- OCISessionBegin()

### 11.4.2.6 Destroy the Connection Pool

`OCIConnectionPoolDestroy()` destroys it.

Use `OCIConnectionPoolDestroy()` to destroy the connection pool.

**See Also:**

OCIConnectionPoolDestroy()

### 11.4.2.7 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((void *) svchp[i], errhp));
 }
 checkerr(errhp, OCIConnectionPoolDestroy(poolhp, errhp, OCI_DEFAULT));
 checkerr(errhp, OCIHandleFree((void *)poolhp, OCI_HTYPE_CPOOL));
```

**See Also:**

- OCIHandleFree()
- Connection Pool Handle Attributes
- OCIConnectionPoolCreate(), OCILogon2(), and OCIConnectionPoolDestroy()

### 11.4.3 Examples of OCI Connection Pooling

Where to find examples of connection pooling in tested complete programs.

Examples of connection pooling in tested complete programs can be found in `cdemocp.c` and `cdemocpproxy.c` in directory `demo`.

## 11.5 When to Use Connection Pooling, Session Pooling, or Neither

Indicates the circumstances in which to use them or not.

If database sessions are not reusable by mid-tier threads (that is, they are stateful) 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 stateful) and the number of back-end server processes is never large enough to potentially cause any scaling issue on the database, there is no need to use any pooling mechanism.

> **Note:**
>
> Having nonpooled sessions or connections results in tearing down and re-creating 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, Oracle strongly recommends that you adopt one of the pooling strategies for mid-tier applications based on whether the database session is stateful 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 must be aware that any kind of pooling works effectively with relatively short tasks. However, if the application is performing a long-running transaction, 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.

Note the following additional information about connection pooling and session pooling:

- OCI Connection Pooling

  Connections to the database are pooled. Sessions are created and destroyed by the user. Each call to the database picks up an appropriate available connection from the pool.

  The application is multiplexing several sessions over fewer physical connections to the database. The users can tune the pool configuration to achieve required concurrency.

  The life-time of the application sessions is independent of the life-time of the cached pooled connections.

- OCI Session Pooling

  Sessions and connections are pooled by OCI. The application gets sessions from the pool and releases sessions back to the pool.

This section includes the following topics:

- Functions for Session Creation
- About Choosing Between Different Types of OCI Sessions

## 11.5.1 Functions for Session Creation

There are a number of ways to create a session with varying functionality.

OCI offers the following functions for session creation:

- `OCILogin`

OCILogon() 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, such as session migration, proxy authentication, or using a connection pool or a session pool.

- `OCILogon2()`

  OCILogon2() 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.

- `OCISessionBegin()`

  OCISessionBegin() 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 must be explicitly allocated and all attributes set. `OCIServerAttach()` must be called before this call.

- `OCISessionGet()`

  OCISessionGet() 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:**

- OCILogon()
- Application Initialization, Connection, and Session Creation
- OCILogon2()
- OCISessionBegin()
- OCISessionGet()

## 11.5.2 About Choosing Between Different Types of OCI Sessions

How to choose the type od session to use.

OCI includes the following types of sessions:

- Basic OCI sessions

  The basic OCI session works by using user name and password over a dedicated OCI server handle. This is the no-pool mechanism. See When to Use Connection Pooling, Session Pooling, or Neither for information of when to use it.

If authentication is obtained through external credentials, then a user name or password is not required.

- Session pool sessions

  Session pool 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 When to Use Connection Pooling, Session Pooling, or Neither on connection pool sessions versus session pooling sessions versus no-pooling sessions.

- Connection pool sessions

  Connection pool sessions are created using `OCISessionGet()` and `OCISessionBegin()` calls from an OCI connection pool. There is no session cache as these are stateful 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 When to Use Connection Pooling, Session Pooling, or Neither 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

  Proxy sessions are useful if the password of the client must be protected from the middle tier. Proxy sessions can also be part of an OCI connection pool or an OCI session pool.

- Migratable Sessions

With transaction handles being migratable, there should be no need for applications to use migratable sessions, instead use OCI connection pooling.

**See Also:**

- OCISessionGet()
- OCISessionRelease()
- OCISessionBegin()
- Middle-Tier Applications in OCI for more information about proxy sessions
- OCI Session Management

# 12

# High Availability in OCI

This chapter describes high availability (HA) features in OCI.

This chapter includes the following topics:

- Runtime Connection Load Balancing
- HA Event Notification
- Transparent Application Failover in OCI
- OCI and Transaction Guard
- OCI and Application Continuity

## 12.1 Runtime Connection Load Balancing

Runtime connection load balancing routes work requests to sessions in a session pool that best serve the work.

It occurs when an application selects a session from an existing session pool and thus is a very frequent activity. For session pools that support services at one instance only, the first available session in the pool is adequate. When the pool supports services that span multiple instances, there is a need to distribute the work requests across instances so that the instances that are providing better service or have greater capacity get more requests.

Applications must connect to an Oracle RAC instance to enable runtime connection load balancing. Furthermore, these applications must:

- Initialize the OCI Environment in `OCI_EVENTS` mode
- Connect to a service that has runtime connection load balancing enabled (use the `DBMS_SERVICE.MODIFY_SERVICE` procedure to set `GOAL` and `CLB_GOAL` as appropriate)
- Link with a thread library

**See Also:**

- *Oracle Real Application Clusters Administration and Deployment Guide* for information about load balancing advisory
- *Oracle Database Development Guide* for information about enabling and disabling runtime connection load balancing for the supported interfaces, and receiving load balancing advisory FAN events

## 12.2 HA Event Notification

Use HA event notification to provide a best-effort programmatic signal to the client if there is a database failure for high availability clients connected to an Oracle RAC database.

Suppose that a 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 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 an Oracle RAC database, you can use HA event notification to provide a best-effort programmatic signal to the client if there is 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 that 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 because 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 is sent to the client, which then disconnects the two connections to instance B and invokes the registered callback. Note that if another instance C of the same database goes down, the client is not notified (because it does 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. Before the mechanism was introduced in Oracle Database 10g Release 2 (10.2), a failure would result in the connection being broken only after the TCP timeout interval expired, which could take minutes. With HA event notification, the 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 is also automatically engaged by OCI.

In the current release, this functionality depends on Oracle Notification Service (ONS). It requires Oracle Clusterware to be installed and configured on the database server for the clients to receive the HA notifications through ONS. All clusterware installations (for example, Oracle Data Guard) should have the same ONS port. There is no client configuration required for ONS.

> **Note:**
>
> The client transparently gets the ONS server information from the database to which it connects. The application administrator can augment or override that information using the deployment configuration file `oraaccess.xml`.

Applications must connect to an Oracle 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.

This section includes the following topics:

- OCIEvent Handle
- OCI Failover for Connection and Session Pools
- OCI Failover for Independent Connections
- Event Callback
- Custom Pooling: Tagged Server Handles
- About Determining Transparent Application Failover (TAF) Capabilities

> **See Also:**
>
> About Client-Side Deployment Parameters Specified in oraaccess.xml for more information about `oraaccess.xml` and details about the parameters under `<events>`, `<fan>` and `<ons>`

## 12.2.1 OCIEvent Handle

The `OCIEvent` handle encapsulates the attributes from the event payload.

OCI implicitly allocates this handle before 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
> - OCIAttrGet()

## 12.2.2 OCI Failover for Connection and Session Pools

A connection pool in an instance of Oracle RAC consists of a pool of connections connected to different instances of Oracle 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 must close the connections: transparent application failover (TAF) occurs immediately, and those connections are reestablished. The connections that are idle and in the free list of the pool must be purged, so that a bad connection is never returned to the user from the pool.

To accommodate custom connection pools, OCI provides a callback function that can be registered on the environment handle. If registered, this callback is invoked when an HA event occurs. Session pools are treated the same way as connection pools. Note that server handles from OCI connection pools or session pools are not passed to the callback. Hence in some cases, the callback could be called with an empty list of connections.

## 12.2.3 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 is 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 are broken out immediately, so that TAF can begin. Note that this applies for the "in-use" connections of connection and session pools also.

## 12.2.4 Event Callback

Shows the signature of the event callback of type `OCIEventCallback`.

The event callback, of type `OCIEventCallback`, has the following signature:

```
void evtcallback_fn (void      *evtctx,
                     OCIEvent  *eventhp );
```

In this signature `evtctx` is the client context, and `OCIEvent` is an event handle that is opaque to the OCI library. The other input argument is `eventhp`, the event handle (the attributes associated with an event).

If registered, this function is called once for each event. For Oracle RAC HA events, this callback is 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 data type `OCIEventCallback *`. It is read-only.
- `OCI_ATTR_EVTCTX` is of data type `void *`. 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, (void *) evtcallback_fn,
           (ub4) 0, (ub4) OCI_ATTR_EVTCBK, errhp);
OCIAttrSet(envhp, (ub4) OCI_HTYPE_ENV, (void *) 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 Oracle 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_SRVFIRST` and `OCI_ATTR_HA_SRVNEXT`:

```
OCIAttrGet(eventhp, OCI_HTYPE_EVENT, (void *)&srvhp, (ub4 *)0,
           OCI_ATTR_HA_SRVFIRST, errhp);
/* or, */
OCIAttrGet(eventhp, OCI_HTYPE_EVENT, (void *)&srvhp, (ub4 *)0,
           OCI_ATTR_HA_SRVNEXT, errhp);
```

When called with attribute `OCI_ATTR_HA_SRVFIRST`, this function retrieves the first server handle in the list of server handles affected. When called with attribute `OCI_ATTR_HA_SRVNEXT`, this function retrieves the next server handle in the list. This function returns `OCI_NO_DATA` and `srvhp` is 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 because 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.

> **See Also:**
>
> OCIAttrGet()

## 12.2.5 Custom Pooling: Tagged Server Handles

Using custom pools, you can retrieve the server handle’s tag information so appropriate cleanup can be performed.

The following features apply to custom pools:

- 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()`.
- 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 returns a pointer to the user memory. A `NULL` pointer is returned for those handles not allocated with extra memory. This attribute is read-only and is of data type `void*`.

> **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 with the `OCIHandleFree()` call on the server handle.

Example 12-1 shows an example of event notification.

**Example 12-1 Event Notification**

```
sword retval;
OCIServer *srvhp;
struct myctx {
   void *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, (void **)&srvhp, OCI_HTYPE_SERVER,
                            (size_t)sizeof(myctx), (void **)&myctxp)
/* handle error */
myctxp->parentConn_myctx = <parent connection reference>;

/* In an event callback function, retrieve the pointer to the user memory */
evtcallback_fn(void *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((void *)srvhp, OCI_HTYPE_SERVER, (void *)&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);
...
  }
...
}
```

**See Also:**

- OCIHandleAlloc()
- OCIHandleFree()

## 12.2.6 About 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 determine if a server handle is TAF-enabled:

```
boolean taf_capable;
...
OCIAttrGet(srvhp, (ub4) OCI_HTYPE_SERVER, (void *) &taf_capable,
           (ub4) sizeof(taf_capable), (ub4)OCI_ATTR_TAF_ENABLED, errhp);
...
```

In this example, `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 an attribute that is a pointer to a Boolean variable and is read-only; `errhp` is an input error handle.

## 12.3 Transparent Application Failover in OCI

Transparent application failover (TAF) is a client-side feature designed to minimize disruptions to end-user applications that occur when database connectivity fails because of instance or network failure.

TAF can be implemented on a variety of system configurations including Oracle Real Application Clusters (Oracle RAC) and Oracle Data Guard physical standby databases. TAF can also be used after restarting a single instance system (for example, when repairs are made).

TAF can be configured to restore database sessions and optionally, to replay open queries. Starting with Oracle Database 10g Release 2 (10.2) all statements that an application attempts to use after a failure attempt failover. That is, an attempt to execute or fetch against other statements engages TAF recovery just as for the failure-time statement. Subsequent statements may now succeed (whereas in the past they failed), or the application may receive errors corresponding to an attempted TAF recovery (such as `ORA-25401`).

> **Note:**
>
> Oracle recommends for applications to register a callback, so when failover happens, the callback can be used to restore the session to the desired state.

> **Note:**
>
> TAF is not supported for remote database links or for DML statements.

This section contains the following topics:

- About Configuring Transparent Application Failover
- Transparent Application Failover Callbacks in OCI
- Transparent Application Failover Callback Structure and Parameters
- Failover Callback Structure and Parameters
- Failover Callback Registration
- Failover Callback Example
- Handling OCI_FO_ERROR

## 12.3.1 About Configuring Transparent Application Failover

TAF can be configured on both the client side and the server side. If both are configured, server-side settings take precedence.

Configure TAF on the client side by including the `FAILOVER_MODE` parameter in the `CONNECT_DATA` portion of a connect descriptor.

Configure TAF on the server side by modifying the target service with the `DBMS_SERVICE.MODIFY_SERVICE` packaged procedure.

An initial attempt at failover may not always succeed. OCI provides a mechanism for retrying failover after an unsuccessful attempt.

**See Also:**

- *Oracle Database Net Services Reference* for more information about client-side configuration of TAF (Connect Data Section)
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the server-side configuration of TAF (DBMS_SERVICE)

## 12.3.2 Transparent Application Failover Callbacks in OCI

Because of the delay that can occur during failover, the application developer may want to inform the user that failover is in progress, and request that the user wait for notification that failover is complete.

Additionally, the session on the initial instance may have received some `ALTER SESSION` commands. These `ALTER SESSION` commands are not automatically replayed on the second instance. Consequently, the developer may want to replay them on the second instance. `OCIAttrSet()` calls that affect the session must also be reexecuted.

To accommodate these requirements, the application developer can register a failover callback function. If failover occurs, the callback function is invoked several times while reestablishing the user's session.

The first call to the callback function occurs when the database 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.

Once the connection has been reestablished, the client may want 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 cannot occur. Additionally, the callback is called each time a user handle besides the primary handle is reauthenticated on the new connection. Because each user handle represents a server-side session, the client may want to replay `ALTER SESSION` commands for that session.

**See Also:**

- OCIAttrSet()
- Handling OCI_FO_ERROR for more information about this scenario

## 12.3.3 Transparent Application Failover Callback Structure and Parameters

Describes the TAF Callback structure and parameters.

The basic structure of a Transparent Application Failover (TAF) callback function is as follows:

```
sb4  TAFcbk_fn(OCISvcCtx *svchp,
               OCIEnv    *envhp,
               void      *fo_ctx,
               ub4        fo_type,
               ub4        fo_event);
```

**svchp**
The service context handle.

**envhp**
The OCI environment handle.

**fo_ctx**
The client context. This is a pointer to memory specified by the client. In this area the client can keep any necessary state or context.

**fo_type**
The failover type. This lets the callback know what type of failover the client has requested. The usual values are as follows:

- `OCI_FO_SESSION` indicates that the user has requested only session failover.
- `OCI_FO_SELECT` indicates that the user has requested select failover as well.

**fo_event**
The failover event indicates the current status of the failover.

- `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 reauthenticated. To determine which one, the application checks the `OCI_ATTR_SESSION` attribute of the service context handle `svchp`.

If Application Continuity is configured, the TAF callback is called with `OCI_FO_END` after successfully re-connecting, re-authenicating, and determining the status of the inflight transaction.

Upon completion of the TAF callback, OCI returns an error if an open transaction is present and Application Continuity for OCI is enabled.

## 12.3.4 Failover Callback Structure and Parameters

Shows and describes the basic structure of a user-defined application failover callback function.

The basic structure of a user-defined application failover callback function is as follows:

```
sb4 appfocallback_fn ( void       * svchp,
                       void       * envhp,
                       void       * fo_ctx,
                       ub4        fo_type,
                       ub4        fo_event );
```

An example is provided in "Failover Callback Example" on page 9-31 for the following parameters:

**svchp**
The first parameter, `svchp`, is the service context handle. It is of type `void *`.

**envhp**
The second parameter, `envhp`, is the OCI environment handle. It is of type `void *`.

**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 `void *`.

**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 as follows:

- `OCI_FO_SESSION` indicates that the user has requested only session failover.
- `OCI_FO_SELECT` 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 reauthenticated. To determine which one, the application checks the `OCI_ATTR_SESSION` attribute of the service context handle (which is the first parameter).

## 12.3.5 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.

> **See Also:**
>
> Example 12-3 for an example of callback registration

## 12.3.6 Failover Callback Example

Shows several failover callback examples.

This section shows an example of a simple user-defined callback function definition (see Example 12-2), failover callback registration (see Example 12-3), and failover callback unregistration (see Example 12-4).

**Example 12-2    User-Defined Failover Callback Function Definition**

```
sb4  callback_fn(svchp, envhp, fo_ctx, fo_type, fo_event)
void * svchp;
void * envhp;
void *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 occur.\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;
}
```

### Example 12-3    Failover Callback Registration

```
int register_callback(srvh, errh)
void *srvh; /* the server handle */
OCIError *errh; /* the error handle */
{
  OCIFocbkStruct failover;                 /*  failover callback structure */
  /* allocate memory for context */
  if (!(failover.fo_ctx = (void *)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,
                (void *) &failover, (ub4) 0,
                (ub4) OCI_ATTR_FOCBK, errh)  != OCI_SUCCESS)
     return(2);
  /* successful conclusion */
  return (0);
}
```

### Example 12-4    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;
  /* unregister the callback */
  status = OCIAttrSet(srvhp, (ub4) OCI_HTYPE_SERVER,
                      (void *) &failover, (ub4) 0,
                      (ub4) OCI_ATTR_FOCBK, errhp);
```

## 12.3.7 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`, however, provides the callback function with the opportunity to handle the error. 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 Database server.
>
> Failover does not work if a LOB column is part of the select list.

Consider the timeline of events presented in Table 12-1.

**Table 12-1 Time and Event**

| Time | Event |
|---|---|
| T0 | Database fails (failure lasts until T5). |
| T1 | Failover is triggered by user activity. |
| T2 | User attempts to reconnect; attempt fails. |
| T3 | Failover callback is invoked with `OCI_FO_ERROR`. |
| T4 | Failover callback enters a predetermined sleep period. |
| T5 | Database comes back up again. |
| T6 | Failover callback triggers a 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.

Example 12-5 shows a callback function that you can use 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:

**Example 12-5 Callback Function That Implements a Failover Strategy**

```
/*--------------------------------------------------------------------*/
/* the user-defined failover callback  */
/*--------------------------------------------------------------------*/
sb4  callback_fn(svchp, envhp, fo_ctx, fo_type, fo_event )
void * svchp;
void * envhp;
void *fo_ctx;
ub4 fo_type;
ub4 fo_event;
```

```
{
   OCIError *errhp;
   OCIHandleAlloc(envhp, (void **)&errhp, (ub4) OCI_HTYPE_ERROR,
              (size_t) 0, (void **) 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 occur.\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;
}
```

## 12.4 OCI and Transaction Guard

Transaction Guard introduces the concept of at-most-once transaction execution in case of a planned or unplanned outage to help prevent an application upon failover from submitting a duplicate submission of an original submission.

When an application opens a connection to the database using this service, the logical transaction ID (LTXID) is generated at authentication and stored in the session handle. This is a globally unique ID that identifies the database transaction from the application perspective. When there is an outage, an application using Transaction Guard can retrieve the LTXID from the previous failed session's handle and use it to determine the outcome of the transaction that was active prior to the session failure. If the LTXID is determined to be unused, then the application can replay an uncommitted transaction by first blocking the original submission using the retrieved LTXID. If the LTXID is determined to be used, then the transaction is committed and the result is returned to the application.

Transaction Guard is a developer API supported for JDBC Type 4 (Oracle Thin), OCI, OCCI, and Oracle Data Provider for .NET (ODP.NET) drivers. For OCI, when an application is written to support Transaction Guard, upon an outage, the OCI client driver acquires and retrieves the LTXID from the previous failed session's handle by calling `OCI_ATTR_GET()` using the `OCI_ATTR_LTXID` session handle attribute.

This section includes the following topic: Developing Applications that Use Transaction Guard.

> **See Also:**
>
> *Oracle Database Development Guide* for information in the chapter about using Transaction Guard in for an overview of Transaction Guard, supported transaction types, transaction types that are not supported, and database configuration information for using Transaction Guard.

## 12.4.1 Developing Applications that Use Transaction Guard

This section describes developing OCI user applications that use Transaction Guard.

See the chapter about using Transaction Guard in *Oracle Database Development Guide* for more detailed information about developing applications using Transaction Guard.

For the third-party or user application to use Transaction Guard in order to be able to fail over a session for OCI, it must include several major steps:

1. Verify if Transparent Application Failover (TAF) is enabled for the connection. Do not attempt to explicitly use Transaction Guard on a TAF-enabled connection, as TAF will automatically check the `LTXID`.
2. On receipt of an error, determine whether the error is a recoverable error - `OCI_ATTR_ERROR_IS_RECOVERABLE` on `OCI_ERROR` handle. If the error is recoverable, then continue to Step 3.

   > **Note:**
   >
   > Do not attempt to use the LTXID to check transaction outcome if the connection has not suffered a recoverable error.

3. Retrieve the LTXID associated with the failed session by using OCI_ATTR_GET() to get the OCI_ATTR_LTXID from the user session handle.
4. Reconnect to the database.

> **Note:**
>
> The new session will have a new LTXID, but you will not need it when checking the status of the original session.

5. Invoke the `DBMS_APP_CONT.GET_LTXID_OUTCOME` PL/SQL procedure with the LTXID obtained from the `OCI_ATTR_GET()` call. The original LTXID of the failed-over session is marked as forced if that LTXID has not been used. The return state tells the driver if the last transaction was `COMMITTED (TRUE/FALSE)` and `USER_CALL_COMPLETED (TRUE/FALSE)`.
6. The application can replay an uncommitted transaction or return the result to the user. If the replay itself incurs an outage, then the LTXID for the replaying session is used for the `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure.

See the following sections for Transaction Guard usage and examples:

- Typical Transaction Guard Usage
- Transaction Guard Examples

### 12.4.1.1 Typical Transaction Guard Usage

Shows typical usage of Transaction Guard using pseudocode.

The following pseudocode shows a typical usage of Transaction Guard:

1. Receive a FAN down event (or recoverable error)
2. FAN aborts the dead session
3. Call `OCIAttrGet()` using the `OCI_ATTR_TAF_ENABLED` attribute on the server handle. If the value is `TRUE`, stop. If the value is `FALSE`, proceed to the next step.
4. If it is a recoverable error, for OCI (`OCI_ATTR_ERROR_IS_RECOVERABLE` on `OCI_ERROR` handle):
   a. Get the last LTXID from the dead session by calling `OCIAttrGet()` using the `OCI_ATTR_LTXID` session handle attribute to retrieve the LTXID associated with the session's handle
   b. Obtain a new session
   c. Call `DBMS_APP_CONT.GET_LTXID_OUTCOME` with the last LTXID to get the return state
5. If the return state is:
   a. `COMMITTED` and `USER_CALL_COMPLETED`

      Then return the result.

   b. `ELSEIF COMMITTED` and `NOT USER_CALL_COMPLETED`

      Then return the result with a warning (with details, such as out binds or row count was not returned).

c. `ELSEIF NOT COMMITTED`

   Resubmit the transaction or series of calls or both, or return error to user.

> **See Also:**
>
> OCIAttrGet()

### 12.4.1.2 Transaction Guard Examples

Shows a Transaction Guard demo program.

Example 12-6 is an OCI Transaction Guard demo program (`cdemotg.c`) that demonstrates:

- Use of the attribute `OCI_ATTR_ERROR_IS_RECOVERABLE`. When an error occurs, the program checks if the error is recoverable.
- Use of the packaged procedure `DBMS_APP_CONT.GET_LTXID_OUTCOME`. If the error is recoverable, the program calls `DBMS_APP_CONT.GET_LTXID_OUTCOME` to determine the status of the active transaction.

If the transaction has not committed, the program re-executes the failed transaction.

> **Note:**
>
> This program does not modify the session state such as NLS parameters, and so forth. Programs that do so may need to reexecute such commands after obtaining a new session from the pool following the error.

**Example 12-6 Transaction Guard Demo Program**

```
*/

#ifndef OCISP_ORACLE
# include <cdemosp.h>
#endif

/* Maximum Number of threads  */
#define MAXTHREAD 1
static ub4 sessMin = 1;
static ub4 sessMax = 9;
static ub4 sessIncr = 2;

static OCIError   *errhp;
static OCIEnv     *envhp;
static OCISPool   *poolhp=(OCISPool *) 0;
static int employeeNum[MAXTHREAD];

static OraText *poolName;
static ub4 poolNameLen;
static CONST OraText *database = (text *)"ltxid_service";
static CONST OraText *appusername =(text *)"scott";
static CONST OraText *apppassword =(text *)"tiger";
static CONST char getLtxid[]=
  ("BEGIN DBMS_APP_CONT.GET_LTXID_OUTCOME ("
   ":ltxid,:committed,:callComplete); END;");

static CONST char insertst1[] =
  ("INSERT INTO EMP(ENAME, EMPNO) values ('NAME1', 1000)");

static void checkerr (OCIError *errhp, sword status);
static void threadFunction (dvoid *arg);

int main(void)
{
  int i = 0;
  sword lstat;
  int timeout =1;
  OCIEnvCreate (&envhp, OCI_THREADED, (dvoid *)0,  NULL,
                NULL, NULL, 0, (dvoid *)0);

  (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                        (size_t) 0, (dvoid **) 0);

  (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &poolhp, OCI_HTYPE_SPOOL,
                        (size_t) 0, (dvoid **) 0);

  /* Create the session pool */
  checkerr(errhp, OCIAttrSet((dvoid *) poolhp,
           (ub4) OCI_HTYPE_SPOOL, (dvoid *) &timeout, (ub4)0,
           OCI_ATTR_SPOOL_TIMEOUT, errhp));

  if (lstat = OCISessionPoolCreate(envhp, errhp,poolhp, (OraText **)&poolName,
              (ub4 *)&poolNameLen, database,
              (ub4)strlen((const char *)database),
              sessMin, sessMax, sessIncr,
              (OraText *)appusername,
              (ub4)strlen((const char *)appusername),
              (OraText *)apppassword,
              (ub4)strlen((const char *)apppassword),
              OCI_SPC_STMTCACHE|OCI_SPC_HOMOGENEOUS))
  {
    checkerr(errhp,lstat);
  }

  printf("Session Pool Created \n");

  /* Multiple threads using the session pool */
  {
    OCIThreadId *thrid[MAXTHREAD];
    OCIThreadHandle *thrhp[MAXTHREAD];

    OCIThreadProcessInit ();
    checkerr (errhp, OCIThreadInit (envhp, errhp));
    for (i = 0; i < MAXTHREAD; ++i)
    {
      checkerr (errhp, OCIThreadIdInit (envhp, errhp, &thrid[i]));
      checkerr (errhp, OCIThreadHndInit (envhp, errhp, &thrhp[i]));
    }
    for (i = 0; i < MAXTHREAD; ++i)
    {
      employeeNum[i]=i;
      /* Inserting into EMP table */
      checkerr (errhp, OCIThreadCreate (envhp, errhp, threadFunction,
                (dvoid *) &employeeNum[i], thrid[i], thrhp[i]));
    }
    for (i = 0; i < MAXTHREAD; ++i)
    {
      checkerr (errhp, OCIThreadJoin (envhp, errhp, thrhp[i]));
      checkerr (errhp, OCIThreadClose (envhp, errhp, thrhp[i]));
      checkerr (errhp, OCIThreadIdDestroy (envhp, errhp, &(thrid[i])));
      checkerr (errhp, OCIThreadHndDestroy (envhp, errhp, &(thrhp[i])));
    }
    checkerr (errhp, OCIThreadTerm (envhp, errhp));
  } /* ALL THE THREADS ARE COMPLETE */
  lstat =  OCISessionPoolDestroy(poolhp, errhp, OCI_DEFAULT);

  printf("Session Pool Destroyed \n");

  if (lstat != OCI_SUCCESS)
    checkerr(errhp, lstat);

  checkerr(errhp, OCIHandleFree((dvoid *)poolhp, OCI_HTYPE_SPOOL));

  checkerr(errhp, OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR));
  return 0;

} /* end of main () */

/* Inserts records into EMP table */
static void threadFunction (dvoid *arg)
{
  int empno = *(int *)arg;
  OCISvcCtx *svchp = (OCISvcCtx *) 0;
  OCISvcCtx *svchp2 = (OCISvcCtx *) 0;
  OCISession *embUsrhp = (OCISession *)0;
  OCIBind *bnd1p, *bnd2p, *bnd3p;

  OCIStmt *stmthp = (OCIStmt *)0;
  OCIStmt *getLtxidStm = (OCIStmt *)0;
  OCIError  *errhp2 = (OCIError *) 0;
  OCIAuthInfo *authp = (OCIAuthInfo *)0;
  sword lstat;
  text name[10];

  boolean callCompl, committed, isRecoverable;
  ub1 *myLtxid;
  ub4  myLtxidLen;

  ub4 numAttempts = 0;

  (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp2, OCI_HTYPE_ERROR,
                   (size_t) 0, (dvoid **) 0);

  lstat =  OCIHandleAlloc((dvoid *) envhp,
                        (dvoid **)&authp, (ub4) OCI_HTYPE_AUTHINFO,
                        (size_t) 0, (dvoid **) 0);
  if (lstat)
    checkerr(errhp2, lstat);

  checkerr(errhp2, OCIAttrSet((dvoid *) authp,(ub4) OCI_HTYPE_AUTHINFO,
           (dvoid *) appusername, (ub4) strlen((char *)appusername),
           (ub4) OCI_ATTR_USERNAME, errhp2));
  checkerr(errhp2,OCIAttrSet((dvoid *) authp,(ub4) OCI_HTYPE_AUTHINFO,
           (dvoid *) apppassword, (ub4) strlen((char *)apppassword),
           (ub4) OCI_ATTR_PASSWORD, errhp2));

restart:
  if  (lstat = OCISessionGet(envhp, errhp2, &svchp, authp,
               (OraText *)poolName, (ub4)strlen((char *)poolName), NULL,
               0, NULL, NULL, NULL, OCI_SESSGET_SPOOL))
  {
    checkerr(errhp2,lstat);
  }

  /* save the ltxid from the session in case we need to call
   *  get_ltxid_outcome to determine the transaction status.
   */
  checkerr(errhp2, OCIAttrGet(svchp, OCI_HTYPE_SVCCTX,
                              (dvoid *)&embUsrhp, (ub4 *)0,
                              (ub4)OCI_ATTR_SESSION, errhp2));
  checkerr(errhp2, OCIAttrGet(embUsrhp, OCI_HTYPE_SESSION,
                              (dvoid *)&myLtxid, (ub4 *)&myLtxidLen,
                              (ub4)OCI_ATTR_LTXID, errhp2));


  /* */
  checkerr(errhp2, OCIStmtPrepare2(svchp, &stmthp, errhp2,
                                   (CONST OraText *)insertst1,
                                   (ub4)sizeof(insertst1),
                                   (const oratext *)0, (ub4)0,
                                   OCI_NTV_SYNTAX, OCI_DEFAULT));

  if (!numAttempts)
  {
    char input[1];

    printf("Kill SCOTT's session now. Press ENTER when complete\n");
    gets(input);
  }
  lstat = OCIStmtExecute (svchp, stmthp, errhp2, (ub4)1, (ub4)0,
                          (OCISnapshot *)0, (OCISnapshot *)0,
                          OCI_DEFAULT );
  if (lstat == OCI_ERROR)
  {
    checkerr(errhp2, OCIAttrGet(errhp2, OCI_HTYPE_ERROR,
                                (dvoid *)&isRecoverable, (ub4 *)0,
                                (ub4)OCI_ATTR_ERROR_IS_RECOVERABLE, errhp2));
    if (isRecoverable)
    {

      printf("Recoverable error occurred; checking transaction status.\n");
      /* get another session to use for the get_ltxid_outcome call */
      if  (lstat = OCISessionGet(envhp, errhp2, &svchp2, authp,
                                 (OraText *)poolName,
                                 (ub4)strlen((char *)poolName), NULL,
                                 0, NULL, NULL, NULL, OCI_SESSGET_SPOOL))
      {
        checkerr(errhp2,lstat);
      }

      checkerr(errhp2,OCIStmtPrepare2(svchp2,&getLtxidStm, errhp2,
                                      (CONST OraText *)getLtxid,
                                      (ub4)sizeof(getLtxid),
                                      (const oratext *)0, (ub4)0,
                                      OCI_NTV_SYNTAX, OCI_DEFAULT));
      checkerr(errhp, OCIBindByPos(getLtxidStm, &bnd1p, errhp, 1,
                                   (dvoid *) myLtxid, (sword)myLtxidLen,
                                   SQLT_BIN, (dvoid *)0,
                                   (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0,
                                   OCI_DEFAULT));
      checkerr(errhp, OCIBindByPos(getLtxidStm, &bnd2p, errhp, 2,
                                   (dvoid *) &committed,
                                   (sword)sizeof(committed),
                                   SQLT_BOL, (dvoid *)0,
                                   (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0,
                                   OCI_DEFAULT));
      checkerr(errhp, OCIBindByPos(getLtxidStm, &bnd3p, errhp, 3,
                                   (dvoid *) &callCompl,
                                   (sword)sizeof(callCompl),
                                   SQLT_BOL, (dvoid *)0,
                                   (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0,
                                   OCI_DEFAULT));

      checkerr(errhp2,OCIStmtExecute(svchp2, getLtxidStm, errhp2,
                                     (ub4)1, (ub4)0,
                                     (OCISnapshot *)0, (OCISnapshot *)0,
                                     OCI_DEFAULT ));
      checkerr(errhp2, OCISessionRelease(svchp2, errhp2,
                                         NULL, 0, OCI_DEFAULT));
      if (committed && callCompl)
        printf("Insert successfully commited \n");
      else if (!committed)
      {
        printf("Transaction did not commit; re-executing last transaction\n");
        numAttempts++;

        /* As there was an outage, do not return this session to the pool */
        checkerr(errhp2,
                 OCISessionRelease(svchp, errhp2,
                                   NULL, 0, OCI_SESSRLS_DROPSESS));
        svchp = (OCISvcCtx *)0;
        goto restart;
      }
    }
  }
  else
  {
    checkerr(errhp2, OCITransCommit(svchp,errhp2,(ub4)0));
    printf("Transaction committed successfully\n");
  }
  if (stmthp)
    checkerr(errhp2, OCIStmtRelease((dvoid *) stmthp, errhp2,
                                    (void *)0, 0, OCI_DEFAULT));
  if (getLtxidStm)
    checkerr(errhp2, OCIStmtRelease((dvoid *) getLtxidStm, errhp2,
                                    (void *)0, 0, OCI_DEFAULT));

  if (svchp)
    checkerr(errhp2, OCISessionRelease(svchp, errhp2, NULL, 0, OCI_DEFAULT));
  OCIHandleFree((dvoid *)authp, OCI_HTYPE_AUTHINFO);
  OCIHandleFree((dvoid *)errhp2, OCI_HTYPE_ERROR);

} /* end of threadFunction (dvoid *) */
```

```
/* This function prints the error */
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;
  }
}
```

## 12.5 OCI and Application Continuity

Application Continuity (AC) gives High Availability (HA) during planned and unplanned outages. OCI support for AC was introduced with Oracle Database 12*c* Release 2 (12.2).

AC masks hardware, software, network, storage errors, and timeouts in a HA environment running either Oracle RAC, Oracle RAC One, or Active Data Guard for instance or site failover. AC provides support for SQL*Plus, Tuxedo, WebLogic Server, and JDBC Type 4 (Oracle Thin), OCI, and Oracle Data Provider for .NET (ODP.NET) drivers.

With planned outages for applications that use the OCI session pool, the OCI session pool detects when a connection has been affected by a `PLANNED DOWN` event and terminates the connection when it is returned to the pool. In planned outages for applications that do not use the OCI session pool, an OCI application detects when a connection has been impacted by a planned shutdown event. In either case, OCI implicitly determines when DML replay is safe and applications see fewer errors following a shutdown event.

With unplanned outages, OCI uses Transaction Guard, which enables an OCI application to reliably determine the outcome of a transaction by recovering an in-flight transaction after a recoverable error occurs. This support means the completion of application requests during outages incurs only a minor delay while restoring database connectivity and session state. AC only attempts to replay an in-flight transaction if it can determine the transaction did not commit during original execution.

For AC support for OCI, Oracle recommends you use an OCI session pool or Tuxedo.

This section includes the following topics:

- About Added Support for Application Continuity
- What Happens Following a Recoverable Error
- Criteria for Successful Replay
- What Factors Disable Application Continuity in OCI
- Failed Replay
- When Is Application Continuity Most Effective

**See Also:**

- *Oracle Real Application Clusters Administration and Deployment Guide* for information about creating services for Application Continuity and Transaction Guard.

## 12.5.1 About Added Support for Application Continuity

Oracle Database release 18c, version 18.1 adds more support for Application Continuity.

Beginning with Oracle Database release 18c, version 18.1, the following support is added for Application Continuity:

- Support is added for OCI dynamic binds and defines for numeric, character, and date/time types. This means the following OCI APIs are extended to support Application Continuity: `OCIBindDynamic()` and `OCIDefineDynamic()`.
- Support is added for binding and defining objects. This means the following OCI APIs are extended to support Application Continuity: `OCIBindObject()`, `OCIDefineObject()`, and `OCITypeByName()`.
- During execution of LOB calls, Application Continuity now supports the handling of connection failure by restarting LOB calls that were interrupted by an outage.
- OCI now supports the new Application Continuity `FAILOVER_TYPE` of `AUTO`, which only attempts to fail over if the session state is known to be restorable at the explicit request boundary.

## 12.5.2 What Happens Following a Recoverable Error

Following a recoverable error, database sessions fail over from one database instance to another database instance.

The new instance may be part of the same Oracle RAC cluster, or an Oracle Data Guard standby database that has been brought up as a primary database following a site failure. After transparent application failover (TAF) successfully reconnects and reauthenticates, Application Continuity in OCI replays the call history associated with the failed session, including all SQL and PL/SQL statements. Replay operates on a single session and does not attempt to synchronize the re-submission activity with any other database session. Replay is successful only if the client-visible results of transaction replay are identical to the original submission.

## 12.5.3 Criteria for Successful Replay

Successful driver replay requires that the client-visible effects of a post-failover transaction be identical to the initial submission.

This success is indicated by the following criteria:

- Return codes and error message text must be identical.
- Result sets must be identical. The define data must be identical and the rows must be returned in the same order.
- The rows processed count must be identical. For example, a post-failover update statement must update the same number of rows as the original update statement.
- Session state for the new connection matches session state from the original connection.

See *Oracle Real Application Clusters Administration and Deployment Guide* for information about these criteria.

### Stability of Mutable Data and Application Continuity

When values change from one execution to the next for a mutable object, its data is considered to be mutable and is thus guaranteed to be non-replayable. Sequences are an example of this mutable data.

To improve the success rate for DML replay, it is necessary to replay DML involving mutable data with the values used at initial submission. If the original values are not kept and if different values for these mutable objects are returned to the client, replay is rejected because the client sees different results.

Support for keeping mutable object values is currently provided for `SYSDATE`, `SYSTIMESTAMP`, `SYS_GUID`, and *`sequence`*`.NEXTVAL`.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* for more information about mutable objects and Application Continuity.

## 12.5.4 What Factors Disable Application Continuity in OCI

Lists the factors that implicitly disables Application Continuity in OCI until the start of the next application request.

The following situations implicitly disables Application Continuity in OCI until the start of the next application request:

- The server detects a condition that is not consistent with replay. For example, for `SESSION_STATE_CONSISTENCY=DYNAMIC` if a PL/SQL anonymous block has an embedded top level `COMMIT` statement (autonomous transactions are not considered top level), the driver implicitly disables Application Continuity in OCI.
- The application calls an OCI function that is not supported by Application Continuity in OCI.

The application can explicitly disable Application Continuity in OCI by calling `OCIRequestDisableReplay()`.

## 12.5.5 Failed Replay

What causes replay to fail.

When Application Continuity in OCI replays a transaction, the following situations will cause replay to fail:

- Encountering a `COMMIT` statement at replay time
- Replay results are not consistent with the initial submission of the transaction
- Presence of a recoverable error during replay if the internal replay retries limit is exceeded
- Applications that use `OCIStmtPrepare()` return the following error: `Error - ORA-25412: transaction replay disabled by call to OCIStmtPrepare`. Use the `OCIStmtPrepare2()` call to support the use of Application Continuity in an HA infrastructure.

Application Continuity returns an error if it cannot successfully replay a failed transaction. Additional diagnostic information will be logged in the client-side trace file to indicate the reason for the replay failure.

## 12.5.6 When Is Application Continuity Most Effective

What determines the effectiveness of Application Continuity in OCI.

Application Continuity in OCI is most effective under the following conditions:

- The database service specifies the `COMMIT_OUTCOME` attribute and transparent application failover (TAF) is configured.
- An application is able to mark the beginning and end of an application request, either explicitly (calling `OCIRequestBegin()` and `OCIRequestEnd()`) or implicitly through use of an OCI session pool.
- An application request contains at most one database transaction that is committed at the end of the request.
- If the application executes PL/SQL or Java in the server, that PL/SQL or Java:
    - Does not have embedded `COMMIT` statements
    - Does not set any state (for example, package variables) that is expected to persist after the PL/SQL or Java completes.
- The TAF callback does not leave an open database transaction.

This section includes the following topics:

- When Application Continuity in OCI Can Fail Over
- Application Continuity in OCI Does Not Support These Constructs
- Possible Side Effects of Application Continuity

## 12.5.6.1 When Application Continuity in OCI Can Fail Over

Describes with which functions when Application Continuity in OCI can fail over if an outage occurs.

Application Continuity in OCI can fail over if an outage occurs during one of the following functions:

- `OCILobAppend()`
- `OCILobArrayRead()`
- `OCILobArrayWrite()`
- `OCILobAssign()`
- `OCILobCharSetForm()`
- `OCILobClose()`
- `OCILobCopy2()`
- `OCILobCreateTemporary()`
- `OCILobDisableBuffering()`
- `OCILobEnableBuffering()`
- `OCILobFileClose()`
- `OCILobFileCloseAll()`
- `OCILobFileGetName()`
- `OCILobFileIsOpen()`
- `OCILobFileOpen()`
- `OCILobFileSetName()`
- `OCILobFlushBuffer()`
- `OCILobFreeTemporary()`
- `OCILobGetChunkSize()`
- `OCILobGetLength()`
- `OCILobGetLength2()`
- `OCILobGetStorageLimit()`
- `OCILobIsEqual()`
- `OCILobIsOpen()`
- `OCILobIsTemporary()`
- `OCILobLoadFromFile()`
- `OCILobLoadFromFile2()`

- `OCILobLocatorAssign()`
- `OCILobLocatorIsInit()`
- `OCILobOpen()`
- `OCILobRead()`
- `OCILobRead2()`
- `OCILobTrim()`
- `OCILobTrim2()`
- `OCILobWriteAppend()`
- `OCILobWriteAppend2()`
- `OCILobWrite()`
- `OCILobWrite2()`
- `OCIPing()`
- `OCIStmtExecute()`
- `OCIStmtFetch()`
- `OCIStmtFetch2()`
- `OCISessionEnd()`
- `OCITransCommit()`
- `OCITransRollback()`

### 12.5.6.2 Application Continuity in OCI Does Not Support These Constructs

What constructs are not supported by Application Continuity in OCI.

Application Continuity in OCI does not support the following constructs:

- XA Transactions
- PL/SQL blocks with embedded `COMMIT` statements
- AQ Dequeue in dequeue immediate mode (deqopt.visibility)
- Streaming binds or defines of descriptor-based types such as objects or lob locators
- Function `OCIStmtPrepare()`
- Registered OCI callbacks of type `OCI_CBTYPE_ENTRY` that do not return `OCI_CONTINUE`
- `COMMIT NOWAIT` statement
- DCL commands

### 12.5.6.3 Possible Side Effects of Application Continuity

Application Continuity in OCI replays the original PL/SQL and SQL statements following a recoverable error once a session is rebuilt and the database state is restored. The replay leaves side-effects that are seen twice, which may or may not be desirable.

It is important that applications understand these side-effects and decide whether duplicate execution is acceptable. If it is not acceptable, then the application must take action to accommodate or mitigate the effects of replay. For example, by calling `OCIRequestDisableReplay()`.

**See Also:**

*Oracle Real Application Clusters Administration and Deployment Guide* for more information about examples of actions that create side effects.

13

# Notification Methods and Database Advanced Queuing

This chapter describes continuous query notification, publish-subscribe notification, and Database advanced queuing features.

This chapter includes the following topics:

- About Continuous Query Notification
- Publish-Subscribe Notification in OCI
- OCI and Database Advanced Queuing

## 13.1 About Continuous Query Notification

Continuous Query Notification (CQN) enables client applications to register queries with the database and receive notifications in response to DML or DDL changes on the objects or in response to result set changes 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 at either the object level or the query level. If registration is at the object level, then whenever a transaction changes any of the registered objects and commits, the notification handler is invoked. If registration is at the query level, then whenever a transaction commits changes such that the result set of the query is modified, the notification handler is invoked, but if the changes do not affect the result set of the query, the notification handler is not invoked.

Query change notification can be registered for the following types of statements: `OCI_STMT_SELECT`, `OCI_STMT_BEGIN`, `OCI_STMT_DECLARE`, and `OCI_STMT_CALL`.

Query change notification assumes that the PLSQL code performs only `SELECT` statements and registers for every `SELECT` statement. Otherwise, it raises an error if there are any non SELECT statements in the PLSQL code.

One use of continuous query notification is in middle-tier applications that must have cached data and keep the cache as recent as possible for the back-end database.

The notification includes the following information:

- Query IDs of queries whose result sets have changed. This is if the registration was at query granularity.
- Names of the modified objects or changed rows.
- Operation type (`INSERT`, `UPDATE`, `DELETE`, `ALTER TABLE`, `DROP TABLE`).

- `ROWID`s of the changed rows and the associated DML operation (`INSERT`, `UPDATE`, `DELETE`).
- Global database events (`STARTUP`, `SHUTDOWN`). In Oracle Real Application Cluster (Oracle RAC) the database delivers a notification when the first instance starts or the last instance shuts down.

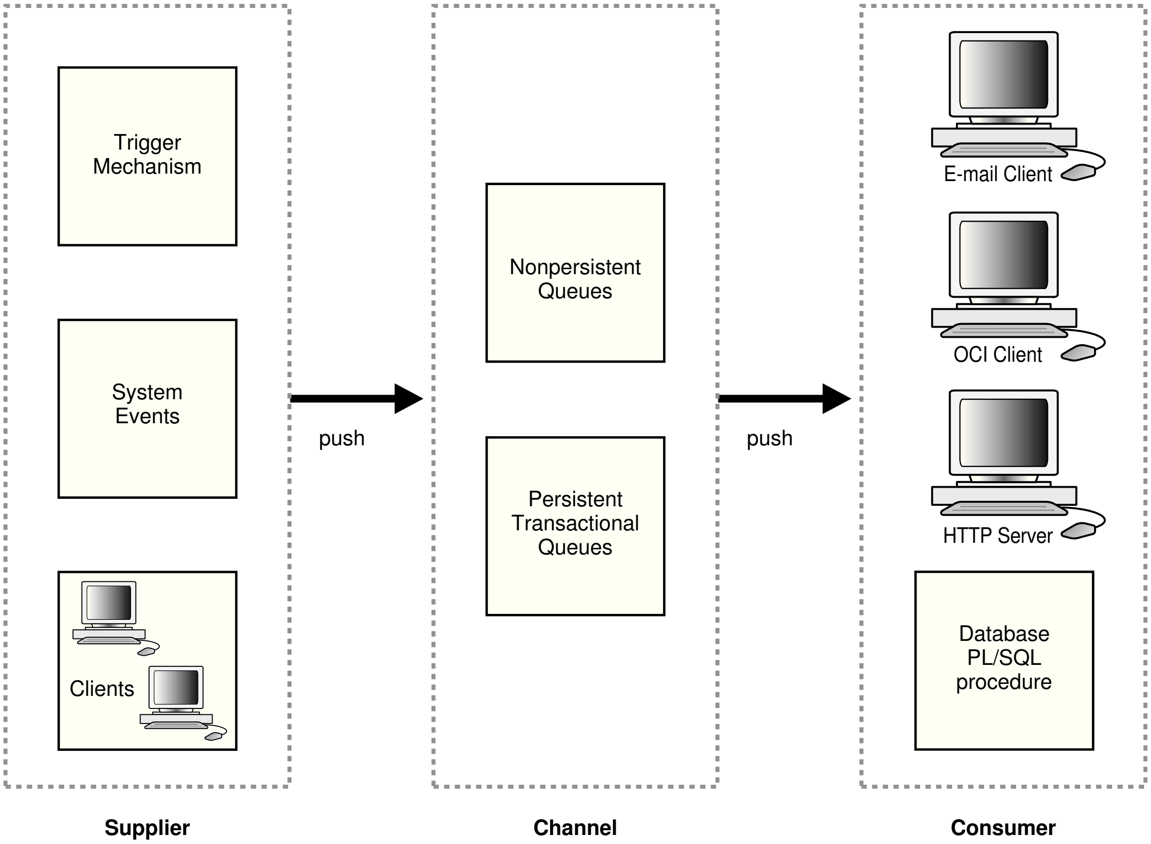
> **See Also:**
>
> - *Oracle Database Development Guide*, the section about using continuous query notification for a complete discussion of the concepts of this feature and using OCI and PL/SQL interfaces to create CQN registrations
> - Publish-Subscribe Notification in OCI

## 13.2 Publish-Subscribe Notification in OCI

The publish-subscribe notification feature allows an OCI application to receive client notifications directly, register an email address to which notifications can be sent, register an HTTP URL to which notifications can be posted, or register a PL/SQL procedure to be invoked on a notification.

Figure 13-1 illustrates the process.

**Figure 13-1    Publish-Subscribe Model**


Trigger Mechanism
System Events
Clients
push
Nonpersistent Queues
Persistent Transactional Queues
push
E-mail Client
OCI Client
HTTP Server
Database PL/SQL procedure
Supplier
Channel
Consumer

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 prespecified email address, or it can be sent to a predefined HTTP URL, or a prespecified database PL/SQL procedure can be invoked because 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.

This section includes the following topics:

- Publish-Subscribe Registration Functions in OCI
- Notification Callback in OCI
- Notification Procedure

- Publish-Subscribe Direct Registration Example
- Publish-Subscribe LDAP Registration Example

> **See Also:**
>
> - OCI and Database Advanced Queuing for information about Database Advanced Queuing
> - *Oracle Database Advanced Queuing User's Guide* for information about creating queues and about Database AQ, including concepts, features, and examples
> - The chapter about `CREATE TRIGGER` in the *Oracle Database SQL Language Reference* for information about creating triggers

## 13.2.1 Publish-Subscribe Registration Functions in OCI

You can register directly to the database or register using Lightweight Directory Access Protocol (LDAP).

Registration can be done in two ways:

- Direct registration. You register directly to the database. This way is simple and the registration takes effect immediately.
- Open registration. You register using Lightweight Directory Access Protocol (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 simultaneously.

This section includes the following topics:

- Publish-Subscribe Register Directly to the Database
- Open Registration for Publish-Subscribe
- Using OCI to Open Register with LDAP
- Setting QOS, Timeout Interval, Namespace, Client Address, and Port Number
- OCI Functions Used to Manage Publish-Subscribe Notification

> **See Also:**
>
> - Publish-Subscribe Register Directly to the Database
> - Open Registration for Publish-Subscribe

### 13.2.1.1 Publish-Subscribe Register Directly to the Database

The following steps are required in an OCI application to register directly 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 later.

**See Also:**

- Database Advanced Queuing and Publish-Subscribe Functions
- Publish-Subscribe Direct Registration Example for examples of the use of these functions in an application

**Note:**

The publish-subscribe feature is only available on multithreaded operating systems.

1. Call `OCIEnvCreate()` or `OCIEnvNlsCreate()` 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_HOSTADDR` - Environment handle attribute that sets the client IP (in either IPv4 or IPv6 format) to which notification is sent

      Oracle Database components and utilities support Internet Protocol version 6 (IPv6) addresses.

      **See Also:**

      OCI_ATTR_SUBSCR_HOSTADDR, OCI_ATTR_SUBSCR_IPADDR, and *Oracle Database Net Services Administrator's Guide* for more information about the IPv6 format for IP addresses

    - `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 with the following values:
  - 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. You can use surviving instances of an Oracle RAC database 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.
- `OCI_ATTR_SUBSCR_TIMEOUT` - Registration timeout interval in seconds. The default is 0 if a timeout is not set.
- `OCI_ATTR_SUBSCR_NTFN_GROUPING_CLASS` - notification grouping class

  Notifications can be spaced out by using the grouping NTFN option with the following constants. A value supported for notification grouping class is:

  ```
  #define OCI_SUBSCR_NTFN_GROUPING_CLASS_TIME   1 /* time  */
  ```
- `OCI_ATTR_SUBSCR_NTFN_GROUPING_VALUE` - notification grouping value in seconds
- `OCI_ATTR_SUBSCR_NTFN_GROUPING_TYPE` - notification grouping type

  Supported values for notification grouping type:

  ```
  #define OCI_SUBSCR_NTFN_GROUPING_TYPE_SUMMARY 1  /* summary */
  #define OCI_SUBSCR_NTFN_GROUPING_TYPE_LAST    2  /* last */
  ```
- `OCI_ATTR_SUBSCR_NTFN_GROUPING_START_TIME` - notification grouping start time
- `OCI_ATTR_SUBSCR_NTFN_GROUPING_REPEAT_COUNT` - notification grouping repeat count

`OCI_ATTR_SUBSCR_NAME`, `OCI_ATTR_SUBSCR_NAMESPACE`, and `OCI_ATTR_SUBSCR_RECPTPROTO` must be set before you register 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 must 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 must be set.

Setting `OCI_ATTR_SUBSCR_CALLBACK` and `OCI_ATTR_SUBSCR_RECPT` at the same time causes an application error.

`OCI_ATTR_SUBSCR_PAYLOAD` is required before the application can perform a post to a subscription.

**See Also:**

Subscription Handle Attributes and About Creating the OCI Environment for setting up the environment with `mode = OCI_EVENTS | OCI_OBJECT`. `OCI_OBJECT` is required for grouping notifications.

4. Set he values of QOS, timeout interval, namespace, and port (see Example 9–15).

**See Also:**

Setting QOS, Timeout Interval, Namespace, Client Address, and Port Number

5. Set `OCI_ATTR_SUBSCR_RECPTPROTO` to `OCI_SUBSCR_PROTO_OCI`, then define the callback routine to be used with the subscription handle.

**See Also:**

Notification Callback in OCI

6. Set `OCI_ATTR_SUBSCR_RECPTPROTO` to `OCI_SUBSCR_PROTO_SERVER`, then define the PL/SQL procedure, to be invoked on notification, in the database.

**See Also:**

Notification Procedure

7. Call `OCISubscriptionRegister()` to register with the subscriptions. This call can register interest in several subscriptions at the same time.

Example 13-1 shows an example of setting QOS levels.

**Example 13-1 Setting QOS Levels, the Notification Grouping Class, Value, and Type, and the Namespace Specific Context**

```
/* Set QOS levels */
ub4 qosflags = OCI_SUBSCR_QOS_PAYLOAD;

/* Set QOS flags in subscription handle */
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                 (dvoid *) &qosflags, (ub4) 0,
                 (ub4) OCI_ATTR_SUBSCR_QOSFLAGS, errhp);

/* Set notification grouping class */
ub4 ntfn_grouping_class = OCI_SUBSCR_NTFN_GROUPING_CLASS_TIME;
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                 (dvoid *) &ntfn_grouping_class, (ub4) 0,
                 (ub4) OCI_ATTR_SUBSCR_NTFN_GROUPING_CLASS, errhp);

/* Set notification grouping value of 10 minutes */
ub4 ntfn_grouping_value = 600;
```

```
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                  (dvoid *) &ntfn_grouping_value, (ub4) 0,
                  (ub4) OCI_ATTR_SUBSCR_NTFN_GROUPING_VALUE, errhp);

/* Set notification grouping type */
ub4 ntfn_grouping_type = OCI_SUBSCR_NTFN_GROUPING_TYPE_SUMMARY;

/* Set notification grouping type in subscription handle */
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                  (dvoid *) &ntfn_grouping_type, (ub4) 0,
                  (ub4) OCI_ATTR_SUBSCR_NTFN_GROUPING_TYPE, errhp);

/* Set namespace specific context */
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                  (dvoid *) NULL, (ub4) 0,
                  (ub4) OCI_ATTR_SUBSCR_NAMESPACE_CTX, errhp);
```

> **See Also:**
>
> - OCIEnvCreate()
> - OCIEnvNlsCreate()
> - OCIHandleAlloc()
> - OCIAttrSet()
> - OCISubscriptionRegister()

### 13.2.1.2 Open Registration for Publish-Subscribe

Lists the prerequisites for the open registration for publish-subscribe.

Prerequisites for the open registration for publish-subscribe are as follows:

- Registering using LDAP (open registration) requires the client to be an enterprise user.

  > **See Also:**
  >
  > *Oracle Database Enterprise User Security Administrator's Guide*, sections about managing enterprise user security

- The compatibility of the database must be 9.0 or later.
- `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 for open registration using Oracle Enterprise Security Manager (OESM) are:

1. In each enterprise domain, create the enterprise role, `ENTERPRISE_AQ_USER_ROLE`.
2. For each database in the enterprise domain, add the global role `GLOBAL_AQ_USER_ROLE` to the enterprise role `ENTERPRISE_AQ_USER_ROLE`.
3. For each enterprise domain, add the enterprise role `ENTERPRISE_AQ_USER_ROLE` to the 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 the enterprise role `ENTERPRISE_AQ_USER_ROLE`.

### 13.2.1.3 Using OCI to Open Register with LDAP

Shows how to open register using LDAP registration.

1. Call `OCIEnvCreate()` or `OCIEnvNlsCreate()` 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 log in 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 for a complete list of authentication modes

   - `OCI_ATTR_LDAP_CTX`: the administrative context for Oracle Database in the LDAP server
3. Call `OCIHandleAlloc()` with handle type `OCI_HTYPE_SUBSCRIPTION`, to allocate a subscription handle.
4. Call `OCIArrayDescriptorAlloc()` 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 with
    - `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 Setting QOS, Timeout Interval, Namespace, Client Address, and Port Number.
8. Call `OCISubscriptionRegister()` to register the subscriptions. The registration takes effect when the database accesses LDAP to pick up new registrations. The frequency of pickups is determined by the value of `LDAP_REG_SYNC_INTERVAL`.

**See Also:**

- OCIEnvCreate()
- OCIEnvNlsCreate()
- OCIAttrSet()
- OCIHandleAlloc()
- OCIArrayDescriptorAlloc()
- OCISubscriptionRegister()

#### 13.2.1.4 Setting QOS, Timeout Interval, Namespace, Client Address, and Port Number

Shows how to set QOSFLAGS to QOS levels using `OCIAttrSet()`.

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 notification is 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((void *)subscrhp, (ub4)OCI_HTYPE_SUBSCRIPTION,
               (void *)&qosflags, (ub4)0, (ub4)OCI_ATTR_SUBSCR_QOSFLAGS, errhp);

/* Set auto-expiration after 30 seconds */
ub4 timeout = 30;
(void)OCIAttrSet((void *)subscrhp, (ub4)OCI_HTYPE_SUBSCRIPTION,
                 (void *)&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. For 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 can 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 is 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((void *)envhp, (ub4)OCI_HTYPE_ENV, (void *)&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((void *)subhp, (ub4)OCI_HTYPE_ENV, (void *)&port, (ub4)0,
                 (ub4)OCI_ATTR_SUBSCR_PORTNO, errhp);
```

Example to set client address:

```
text ipaddr[16] = "10.177.246.40";
(void)(OCIAttrSet((dvoid *) envhp, (ub4) OCI_HTYPE_ENV,
       (dvoid *) ipaddr, (ub4) strlen((const char *)ipaddr),
       (ub4) OCI_ATTR_SUBSCR_IPADDR, errhp));
```

**See Also:**

OCI_ATTR_SUBSCR_IPADDR

### 13.2.1.5 OCI Functions Used to Manage Publish-Subscribe Notification

Lists and describes the functions used to manage publish-subscribe notification.

Table 13-1 lists the functions that are used to manage publish-subscribe notification.

**Table 13-1 Publish-Subscribe Functions**

| Function | Purpose |
|---|---|
| OCISubscriptionDisable() | Disables a subscription |
| OCISubscriptionEnable() | Enables a subscription |
| OCISubscriptionPost() | Posts a subscription |
| OCISubscriptionRegister() | Registers a subscription |
| OCISubscriptionUnRegister() | Unregisters a subscription |

## 13.2.2 Notification Callback in OCI

The client must 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

The callback must return a value of `OCI_CONTINUE` and adhere to the following specification:

```
typedef ub4 (*OCISubscriptionNotify) ( void            *pCtx,
                                       OCISubscription *pSubscrHp,
                                       void            *pPayload,
                                       ub4             iPayloadLen,
                                       void            *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. Currently, 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 (AQ), the descriptor is `OCI_DTYPE_AQNFY`. For the AQ namespace, the count of notifications received in the group is provided in the notification descriptor. The attributes of `pDescriptor` are:

- Notification flag (regular = 0, timeout = 1, or grouping notification = 2) - `OCI_ATTR_NFY_FLAGS`
- Queue name - `OCI_ATTR_QUEUE_NAME`
- Consumer name - `OCI_ATTR_CONSUMER_NAME`
- Message ID - `OCI_ATTR_NFY_MSGID`
- Message properties - `OCI_ATTR_MSG_PROP`
- Count of notifications received in the group - `OCI_ATTR_AQ_NTFN_GROUPING_COUNT`
- The group, an OCI collection - `OCI_ATTR_AQ_NTFN_GROUPING_MSGID_ARRAY`

> **See Also:**
>
> - OCI and Database Advanced Queuing
> - Notification Descriptor Attributes

**iMode (IN)**
Call-specific mode. The only valid value is `OCI_DEFAULT`. This value executes the default call.

Example 13-2 shows how to use AQ grouping notification attributes in a notification callback.

**Example 13-2 Using AQ Grouping Notification Attributes in an OCI Notification Callback**

```
ub4 notifyCB1(void *ctx, OCISubscription *subscrhp, void *pay, ub4 payl,
              void *desc, ub4 mode)
{
 oratext            *subname;
 ub4                 size;
 OCIColl            *msgid_array = (OCIColl *)0;
 ub4                 msgid_cnt = 0;
 OCIRaw             *msgid;
 void              **msgid_ptr;
 sb4                 num_msgid = 0;
 void               *elemind = (void *)0;
 boolean             exist;
 ub2                 flags;
 oratext            *hexit = (oratext *)"0123456789ABCDEF";
 ub4                 i, j;

 /* get subscription name */
 OCIAttrGet(subscrhp, OCI_HTYPE_SUBSCRIPTION, (void *)&subname, &size,
           OCI_ATTR_SUBSCR_NAME,ctxptr->errhp);

/* print subscripton name */
printf("Got notification for %.*s\n", size, subname);
fflush((FILE *)stdout);

/* get the #ntfns received in this group */
OCIAttrGet(desc, OCI_DTYPE_AQNFY, (void *)&msgid_cnt, &size,
           OCI_ATTR_AQ_NTFN_GROUPING_COUNT, ctxptr->errhp);

/* get the group - collection of msgids */
OCIAttrGet(desc, OCI_DTYPE_AQNFY, (void *)&msgid_array, &size,
           OCI_ATTR_AQ_NTFN_GROUPING_MSGID_ARRAY, ctxptr->errhp);

/* get notification flag - regular, timeout, or grouping notification? */
OCIAttrGet(desc, OCI_DTYPE_AQNFY, (void *)&flags, &size,
           OCI_ATTR_NFY_FLAGS, ctxptr->errhp);

/* print notification flag */
printf("Flag: %d\n", (int)flags);

/* get group (collection) size */
if (msgid_array)
  checkerr(ctxptr->errhp,
       OCICollSize(ctxptr->envhp, ctxptr->errhp,
       CONST OCIColl *) msgid_array, &num_msgid),
       "Inside notifyCB1-OCICollSize");
else
  num_msgid =0;

/* print group size */
printf("Collection size: %d\n", num_msgid);

/* print all msgids in the group */
for(i = 0; i < num_msgid; i++)
{
  ub4  rawSize;                                             /* raw size    */
  ub1 *rawPtr;                                              /* raw pointer */
    /* get msgid from group */
  checkerr(ctxptr->errhp,
       OCICollGetElem(ctxptr->envhp, ctxptr->errhp,
              (OCIColl *) msgid_array, i, &exist,
              (void **)(&msgid_ptr), &elemind),
       "Inside notifyCB1-OCICollGetElem");
  msgid = *msgid_ptr;
  rawSize = OCIRawSize(ctxptr->envhp, msgid);
  rawPtr = OCIRawPtr(ctxptr->envhp, msgid);

  /* print msgid size */
  printf("Msgid size: %d\n", rawSize);

  /* print msgid in hexadecimal format */
  for (j = 0; j < rawSize; j++)
  {                                           /* for each byte in the raw */
    printf("%c", hexit[(rawPtr[j] & 0xf0) >> 4]);
    printf("%c", hexit[(rawPtr[j] & 0x0f)]);
  }
  printf("\n");
}

/* print #ntfns received in group */
```

```
 printf("Notification Count: %d\n", msgid_cnt);
 printf("\n");
 printf("****************************************************************\n");
 fflush((FILE *)stdout);
 return 1;
}
```

## 13.2.3 Notification Procedure

The PL/SQL notification procedure that is invoked when there is some activity on the subscription for which interest has been registered, must be created in the database.

This procedure is typically set through the `OCI_ATTR_SUBSCR_RECPT` attribute of the subscription handle.

> **See Also:**
>
> - Subscription Handle Attributes
> - "Oracle Database AQ PL/SQL Callback" in *Oracle Database PL/SQL Packages and Types Reference* for the PL/SQL procedure specification

## 13.2.4 Publish-Subscribe Direct Registration Example

Shows examples implementing publish subscription notification using direct registration.

Example 13-3 shows how system events, client notification, and Advanced Queuing work together to implement publish subscription notification.

The PL/SQL code in Example 13-3 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. Connect as `pubsub` before running Example 13-3.

After the subscriptions are created, the client must register for notification using callback functions. Example 13-4 shows sample code that performs the necessary steps for registration. The initial steps of allocating and initializing session handles are omitted here for clarity.

If user `IX` logs on to the database, the client is notified by email, and the callback function `notifySnoop` is called. An email notification is sent to the address xyz@company.com and the PL/SQL procedure `plsqlnotifySnoop` is also called in the database.

**Example 13-3 Implementing a Publish Subscription Notification**

```
----------------------------------------------------------
----create queue table for persistent multiple consumers
----------------------------------------------------------
---- 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;
/
```

### Example 13-4    Registering for Notification Using Callback Functions

```
...
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)
    void *ctx;
    OCISubscription *subscrhp;
    void *pay;
    ub4 payl;
    void *desc;
    ub4 mode;
{
    printf("Notification : User ix Logged on\n");
  (void)OCIHandleFree((void *)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 ((void *) 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;
  void * func;
  ub4 recpproto;
  char * recpaddr;
  ub4 recppres;
{
    /* allocate subscription handle */
    (void) OCIHandleAlloc((void *) envhp, (void **)subscrhp,
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (void **) 0);

    /* set subscription name in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) subscriptionName,
        (ub4) strlen((char *)subscriptionName),
        (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

    /* set callback function in handle */
    if (func)
      (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
          (void *) func, (ub4) 0,
          (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

    /* set context in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) 0, (ub4) 0,
       (ub4) OCI_ATTR_SUBSCR_CTX, errhp);
    /* set namespace in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) &namespace, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

    /* set receive with protocol in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) &recpproto, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_RECPTPROTO, errhp);

    /* set recipient address in handle */
    if (recpaddr)
      (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
          (void *) recpaddr, (ub4) strlen(recpaddr),
          (ub4) OCI_ATTR_SUBSCR_RECPT, errhp);

    /* set receive with presentation in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) &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 initializes a Registration Handle */

/* Register for OCI notification */
    initSubscriptionHn(    &subscrhpSnoop,    /* subscription handle*/
    (char*) "PUBSUB.LOGON:SNOOP", /* subscription name */
                                  /*<queue_name>:<agent_name> */
        (void*)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> */
        (void*)0, /* callback function */
        OCI_SUBSCR_PROTO_MAIL, /* receive with protocol */
        (char*)  "xyz@company.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> */
        (void*)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);
}
```

## 13.2.5 Publish-Subscribe LDAP Registration Example

Shows an example that illustrates how to do LDAP registration.

Example 13-5 shows a code fragment that illustrates how to do LDAP registration. Please read all the program comments.

**Example 13-5    LDAP Registration**

```
...

  /* To use the LDAP registration feature, OCI_EVENTS | OCI_EVENTS |OCI_USE_LDAP*/
  /*   must be set in OCIEnvCreate or OCIEnvNlsCreate */
  /*     (Note: OCIInitialize is deprecated): */
  (void) OCIInitialize((ub4) OCI_EVENTS|OCI_OBJECT|OCI_USE_LDAP, (void *)0,
                       (void * (*)(void *, size_t)) 0,
                       (void * (*)(void *, void *, size_t))0,
                       (void (*)(void *, void *)) 0 );

...

  /* set LDAP attributes in the environment handle */

  /* LDAP host name */
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)"yow", 3,
                    OCI_ATTR_LDAP_HOST, (OCIError *)errhp);

  /* LDAP server port */
  ldap_port = 389;
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)&ldap_port,
                    (ub4)0, OCI_ATTR_LDAP_PORT, (OCIError *)errhp);

  /* bind DN of the client, normally the enterprise user name */
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)"cn=orcladmin",
                    12, OCI_ATTR_BIND_DN, (OCIError *)errhp);

  /* password of the client */
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)"welcome",
                    7, OCI_ATTR_LDAP_CRED, (OCIError *)errhp);

  /* authentication method is "simple", username/password authentication */
  ldap_auth = 0x01;
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)&ldap_auth,
                    (ub4)0, OCI_ATTR_LDAP_AUTH, (OCIError *)errhp);

  /* administrative context: this is the DN above cn=oraclecontext */
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)"cn=acme,cn=com",
                    14, OCI_ATTR_LDAP_CTX, (OCIError *)errhp);

...

  /* retrieve the LDAP attributes from the environment handle */

  /* LDAP host */
  (void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&buf,
                    &szp,  OCI_ATTR_LDAP_HOST,  (OCIError *)errhp);

  /* LDAP server port */
  (void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&intval,
                    0,  OCI_ATTR_LDAP_PORT,  (OCIError *)errhp);

  /* client binding DN */
  (void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&buf,
                    &szp,  OCI_ATTR_BIND_DN,  (OCIError *)errhp);

  /* client password */
  (void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&buf,
                    &szp,  OCI_ATTR_LDAP_CRED,  (OCIError *)errhp);

  /* administrative context */
  (void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&buf,
                    &szp,  OCI_ATTR_LDAP_CTX,  (OCIError *)errhp);

  /* client authentication method */
  (void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&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((void *)envhp, (void **)dn,
                         (ub4) OCI_DTYPE_SRVDN, (size_t)0, (void **)0);

  /* now *dn is the server DN descriptor, add the DN of the first database
     that you want to register */
  (void) OCIAttrSet((void *)*dn, (ub4) OCI_DTYPE_SRVDN,
                    (void *)"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((void *)*dn, (ub4) OCI_DTYPE_SRVDN,
                    (void *)"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((void *) *subhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                 (void *) *dn, (ub4)0, (ub4) OCI_ATTR_SERVER_DNS, errhp);

  ...

  /* now you will try to get the server DN information from the subscription
     handle */

  /* first, get the server DN descriptor out */
  (void) OCIAttrGet((void *) *subhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                    (void *)dn, &szp, OCI_ATTR_SERVER_DNS, errhp);

  /* then, get the number of server DNs in the descriptor */
  (void) OCIAttrGet((void *) *dn, (ub4)OCI_DTYPE_SRVDN, (void *)&intval,
                    &szp, (ub4)OCI_ATTR_DN_COUNT, errhp);

  /* allocate an array of char * to hold server DN pointers returned by
     an Oracle database*/
    if (intval)
    {
      arr = (char **)malloc(intval*sizeof(char *));
      (void) OCIAttrGet((void *)*dn, (ub4)OCI_DTYPE_SRVDN, (void *)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. However,
     if mode OCI_REG_LDAPONLY is used, the registration request
     will be directly sent to LDAP. This mode should be used when there is
     more than one server DN in the server DN descriptor or you are sure
     that a database connection is not available.

     In this example, two DNs are entered, so you should use mode
     OCI_REG_LDAPONLY in LDAP registration. */
  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);
}
...
```

## 13.3 OCI and Database Advanced Queuing

OCI provides an interface to the Database Advanced Queuing (Database AQ) feature. Database Advanced Queuing provides message queuing as an integrated part of Oracle Database.

Database AQ provides this functionality by integrating the queuing system with the database, thereby creating a *message-enabled database.* By providing an integrated solution, Database AQ frees application developers to devote their efforts to their specific business logic rather than having to construct a messaging infrastructure.

> **Note:**
>
> To use Database Advanced Queuing, you must be using the Enterprise Edition of Oracle Database.

This section includes the following topics:

- OCI Database Advanced Queuing Functions
- OCI Database Advanced Queuing Descriptors
- Database Advanced Queuing in OCI Versus PL/SQL
- Using Buffered Messaging

> **See Also:**
>
> - *Oracle Database Advanced Queuing User's Guide*
> - *Oracle XML Developer's Kit Programmer's Guide*
> - The description of OCIAQEnq() for example code demonstrating the use of OCI with AQ

## 13.3.1 OCI Database Advanced Queuing Functions

Lists the OCI Database Advanced Queuing functions.

The OCI library includes several functions related to Database Advanced Queuing:

- `OCIAQEnq()`
- `OCIAQDeq()`
- `OCIAQListen()` (Deprecated)
- `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:**

- Database Advanced Queuing and Publish-Subscribe Functions
- OCIAQEnq()
- OCIAQDeq()
- OCIAQListen()
- OCIAQListen2()
- OCIAQEnqArray()
- OCIAQDeqArray()

## 13.3.2 OCI Database Advanced Queuing Descriptors

Lists the OCI Database Advanced Queuing descriptors and shows their usage.

The following descriptors are used by OCI Database Advanced Queuing operations:

- `OCIAQEnqOptions`
- `OCIAQDeqOptions`
- `OCIAQMsgProperties`
- `OCIAQAgent`

You can allocate these descriptors with 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 that can be set or read.

**See Also:**

- Database Advanced Queuing Descriptor Attributes
- OCIDescriptorAlloc()

## 13.3.3 Database Advanced Queuing in OCI Versus PL/SQL

Shows a comparison between functions, parameters, and options for OCI Database Advanced Queuing functions and descriptors, and PL/SQL AQ functions in the DBMS_AQ package.

The following tables compare functions, parameters, and options for OCI Database Advanced Queuing functions and descriptors, and PL/SQL AQ functions in the DBMS_AQ package. Table 13-2 compares AQ functions.

**Table 13-2 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()` |

Table 13-3 compares the parameters for the enqueue functions.

**Table 13-3 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()` requires the following additional parameters: `svch, errh, payload_tdo, payload_ind, and flags`. |

Table 13-4 compares the parameters for the dequeue functions.

**Table 13-4 Dequeue Parameters**

| DBMS_AQ.DEQUEUE Parameter | OCIAQDeq() Parameter |
|---|---|
| `queue_name` | `queue_name` |

**Table 13-4 (Cont.) Dequeue Parameters**

| DBMS_AQ.DEQUEUE Parameter | OCIAQDeq() Parameter |
|---|---|
| `dequeue_options` | `dequeue_options` |
| `message_properties` | `message_properties` |
| `payload` | `payload` |
| `msgid` | `msgid` |
| - | Note: `OCIAQDeq()` requires the following additional parameters: `svch, errh, dequeue_options, message_properties, payload_tdo, payload, payload_ind, and flags`. |

Table 13-5 compares parameters for the listen functions.

**Table 13-5 Listen Parameters**

| DBMS_AQ.LISTEN Parameter | OCIAQListen2() Parameter |
|---|---|
| `agent_list` | `agent_list` |
| `wait` | `wait` |
| `agent` | `agent` |
| `listen_delivery_mode` | `lopts` |
| - | Note: `OCIAQListen2()` requires the following additional parameters: `svchp, errhp, agent_list, num_agents, agent, lmops, and flags`. |

Table 13-6 compares parameters for the array enqueue functions.

**Table 13-6 Array Enqueue Parameters**

| DBMS_AQ.ENQUEUE_ARRAY Parameter | OCIAQEnqArray() Parameter |
|---|---|
| `queue_name` | `queue_name` |
| `enqueue_options` | `enqopt` |
| `array_size` | `iters` |

**Table 13-6 (Cont.) Array Enqueue Parameters**

| DBMS_AQ.ENQUEUE_ARRAY Parameter | OCIAQEnqArray() Parameter |
|---|---|
| `message_properties_array` | `msgprop` |
| `payload_array` | `payload` |
| `msgid_array` | `msgid` |
| - | Note: `OCIAQEnqArray()` requires the following additional parameters: `svch, errh, payload_tdo, payload_ind, ctxp, enqcbfp, and flags`. |

Table 13-7 compares parameters for the array dequeue functions.

**Table 13-7 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()` requires the following additional parameters: `svch, errh, msgprop, payload_tdo, payload_ind, ctxp, deqcbfp, and flags`. |

Table 13-8 compares parameters for the agent attributes.

**Table 13-8 Agent Parameters**

| PL/SQL Agent Parameter | OCIAQAgent Attribute |
|---|---|
| `name` | `OCI_ATTR_AGENT_NAME` |

**Table 13-8 (Cont.) Agent Parameters**

| PL/SQL Agent Parameter | OCIAQAgent Attribute |
|---|---|
| `address` | `OCI_ATTR_AGENT_ADDRESS` |
| `protocol` | `OCI_ATTR_AGENT_PROTOCOL` |

Table 13-9 compares parameters for the message properties.

**Table 13-9 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` |

Table 13-10 compares enqueue option attributes.

### Table 13-10 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`<br>(deprecated) |
| `transformation` | `OCI_ATTR_TRANSFORMATION` |
| `delivery_mode` | `OCI_ATTR_MSG_DELIVERY_MODE` |

Table 13-11 compares dequeue option attributes.

### Table 13-11 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` |
| `correlation` | `OCI_ATTR_CORRELATION` |
| `deq_condition` | `OCI_ATTR_DEQCOND` |
| `transformation` | `OCI_ATTR_TRANSFORMATION` |
| `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` are set to dequeue this message before the first one. An error is not returned 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 the `OCI_ATTR_SEQUENCE` attribute is not set, but the message is not dequeued before the message with relative message ID.

**See Also:**

OCIAQEnq()

## 13.3.4 Using Buffered Messaging

Buffered messaging is a nonpersistent messaging capability within Database AQ that was first available in Oracle Database 10*g* Release 2.

Buffered messages reside in shared memory and can be lost if there is an instance failure. Unlike persistent messages, redo does not get written to disk. Buffered message enqueue and dequeue is much faster than persistent message operations. Because shared memory is limited, buffered messages may have to be spilled to disk. Flow control can 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:

- `OCIAQEnq()`
- `OCIAQDeq()`
- `OCIAQListen2()`

Example 13-6 shows an example of enqueue buffered messaging.

Example 13-7 shows an example of dequeue buffered messaging.

**Note:**

Array operations are not supported for buffered messaging. Applications can use the `OCIAQEnqArray()` and `OCIAQDeqArray()` functions with the array size set to 1.

### Example 13-6 Enqueue Buffered Messaging

```
...
OCIAQMsgProperties  *msgprop;
OCIAQEnqueueOptions *enqopt;
message              msg;    /* message is an object type */
null_message         nmsg;   /* message indicator */
...
/* Allocate descriptors */
  OCIDescriptorAlloc(envhp, (void **)&enqopt, OCI_DTYPE_AQENQ_OPTIONS, 0,
                     (void **)0));

 OCIDescriptorAlloc(envhp, (void **)&msgprop,OCI_DTYPE_AQMSG_PROPERTIES, 0,
                    (void **)0));

/* Set delivery mode to buffered */
 dlvm = OCI_MSG_BUFFERED;
 OCIAttrSet(enqopt,  OCI_DTYPE_AQENQ_OPTIONS, (void *)&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,(void *)&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, (void **)&mesg,
         (void **)&nmesg, (OCIRaw **)0, 0));
...
```

### Example 13-7 Dequeue Buffered Messaging

```
...
OCIAQMsgProperties  *msgprop;
OCIAQDequeueOptions *deqopt;
...
OCIDescriptorAlloc(envhp, (void **)&mprop, OCI_DTYPE_AQMSG_PROPERTIES, 0,
                   (void **)0));
OCIDescriptorAlloc(envhp, (void **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
                   (void **)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,(void *)&vis, sizeof(ub4),
           OCI_ATTR_VISIBILITY, errhp)
/* delivery mode is buffered */
dlvm  = OCI_MSG_BUFFERED;
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (void *)&dlvm,  sizeof(ub2),
           OCI_ATTR_MSG_DELIVERY_MODE, errhp);
/* Set the consumer for which to dequeue the message (this must be specified
   regardless of the type of message being dequeued).
*/
consumer = "FIRST_SUBSCRIBER";
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (void *)consumer,
           (ub4)strlen((char*)consumer), OCI_ATTR_CONSUMER_NAME, errhp);
/* Dequeue the message but do not return the payload (to simplify the code
   fragment)
*/
```

```
OCIAQDeq(svchp, errhp,  "test_queue", deqopt, msgprop, msg_tdo, (void **)0,
         (void **)0, (OCIRaw**)0, 0);
...
```

**See Also:**

- OCIAQEnq()
- OCIAQDeq()
- OCIAQListen2()

# 14

# User-Defined Callback Functions in OCI

Oracle Call Interface can execute user-specific code in addition to OCI calls.

You can use this functionality for:

- Adding tracing and performance measurement code to enable users to tune their applications
- Performing preprocessing or postprocessing 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 provides support for calling user code before or after executing the OCI calls. It also allows the user-defined code to be executed instead of executing the OCI code.

The user callback code can 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.

This section includes the following topics:

- About Registering User Callbacks in OCI
- OCI Callbacks from External Procedures

> **See Also:**
>
> OCIEnvCreate()

## 14.1 About 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.

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`, then subsequent replacement callbacks and the OCI code do 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.

This section includes the following topics:

- OCIUserCallbackRegister
- User Callback Function
- User Callback Control Flow
- User Callback for OCIErrorGet()
- Errors from Entry Callbacks
- Dynamic Callback Registrations
- About Loading Multiple Packages
- Package Format
- User Callback Chaining
- About Accessing Other Data Sources Through OCI
- Restrictions on Callback Functions
- Example of OCI Callbacks

**See Also:**

OCIUserCallbackGet() and OCIUserCallbackRegister()

## 14.1.1 OCIUserCallbackRegister

A user callback is registered using the OCIUserCallbackRegister() call.

Currently, `OCIUserCallbackRegister()` is only registered on the environment handle. The user's callback function of typedef `OCIUserCallback` is registered along with its context for the OCI call identified by the OCI function code, *fcode*. The type of the callback, whether entry, replacement, or exit, is specified by the *when* parameter.

For example, the `stmtprep_entry_dyncbk_fn` entry callback function and its context `dynamic_context`, are registered against the environment handle `hndlp` for the `OCIStmtPrepare2()` 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);
```

**See Also:**

- OCIUserCallbackRegister()
- OCIStmtPrepare2()

## 14.1.2 User Callback Function

Details about the user callback function.

The user callback function must use the following syntax:

```
typedef sword (*OCIUserCallback)
     (void *ctxp,      /* context for the user callback*/
      void *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, however, the callback returns any return code other than `OCI_CONTINUE`, the last returned return code becomes the return code for the call. At this point, the value returned for `*errnop` 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 are 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.

A null value can be registered to deregister a callback. That is, if the value of the callback (`OCIUserCallback()`) is `NULL` in the `OCIUserCallbackRegister()` call, then the user callback is deregistered.

When using the thread-safe mode, the OCI program acquires all mutexes before calling the user callbacks.

**See Also:**

- OCI Demonstration Programs
- OCIUserCallbackRegister()
- OCIHandleAlloc()

## 14.1.3 User Callback Control Flow

Shows the control flow for a user callback.

Example 14-1 shows pseudocode that describes the overall processing of a typical OCI call.

**Example 14-1 Pseudocode That 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
}
```

## 14.1.4 User Callback 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 `errcodep` parameters of the replacement callback of the `OCIErrorGet()` call.

If, however, the callbacks are either partially overriding OCI code, or just doing some other postprocessing, then they can use the exit callback to modify the error text and `errcodep` parameters of the `OCIErrorGet()` call 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.

**See Also:**

OCIErrorGet()

## 14.1.5 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.

## 14.1.6 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 nonintrusively.

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 deregister 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.

Setting an operating system environment variable, `ORA_OCI_UCBPKG`, controls the loading of the package. This variable names the packages in a generic way. The packages must be located in the `$ORACLE_HOME/lib` directory.

## 14.1.7 About 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, in the past the package was specified as:

```
setenv ORA_OCI_UCBPKG mypkg
```

Currently, you can still specify the package as before, 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`, OCIEnvCreate() or OCIEnvNlsCreate() 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 about user-defined callbacks, see your operating system-specific documentation on compiling and linking applications.

## 14.1.8 Package Format

The package source must provide two functions.

In the past, a package had to specify the source code for the `OCIEnvCallback()` function. However, the `OCIEnvCallback()` function is obsolete. Instead, the package source must provide two functions. The first function must 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)
      void *     metaCtx;         /* The metacontext */
      void *     libCtx;          /* The context for this package. */
      ub4        argfmt;          /* package argument format */
      sword      argc;            /* package arg count*/
      void *     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 is named *packagename* suffixed with the word *EnvCallback*.

This function is a replacement for `OCIEnvCallback()`. Currently, 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, void *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 `usrmempp` 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.

## 14.1.9 User Callback Chaining

User callbacks can 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.

The `OCIUserCallbackGet()` function determines which function and context is registered for an OCI call.

**See Also:**

OCIUserCallbackGet()

## 14.1.10 About Accessing Other Data Sources Through OCI

Because Oracle Database is the predominant database software 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. 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 to access Oracle data sources, and incurs the same penalty that ODBC does for non-Oracle data sources.

## 14.1.11 Restrictions on Callback Functions

Details the 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()`, and `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 the `OCIUserCallbackRegister()` call itself, or for any of the following calls:
  - `OCIUserCallbackGet()`
  - `OCIEnvCreate()`
  - `OCIInitialize()` (Deprecated)
  - `OCIEnvNlsCreate()`

> **See Also:**
>
> - OCIUserCallbackRegister()
> - OCIUserCallbackGet()
> - OCIHandleAlloc()
> - OCIHandleFree()
> - OCILogoff()
> - OCIEnvCreate()
> - OCIInitialize()
> - OCIEnvNlsCreate()

## 14.1.12 Example of OCI Callbacks

Shows examples of using OCI callbacks.

Suppose that there are five packages each registering entry, replacement, and exit callbacks for the `OCIStmtPrepare2()` call. That is, the `ORA_OCI_UCBPKG` variable is set as shown in Example 14-2.

In each package `pkgN` (where N can be 1 through 5), the `pkgNInit()` and `PkgNEnvCallback()` functions are specified, as shown in Example 14-3.

Example 14-4 shows how the `pkgNEnvCallback()` function registers the entry, replacement, and exit callbacks.

Finally, Example 14-5 shows how in the source code for the application, user callbacks can be registered with the `NULL` *ucbDesc*.

Example 14-6 shows that when the `OCIStmtPrepare2()` call is executed, the callbacks are called in the following order.

> **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 `OCIStmtPrepare2()` call are bypassed. Instead, `pkg5_exit_callback_fn()` is executed next.

#### Example 14-2 Environment Variable Setting for the ORA_OCI_UCBPKG Variable

```
setenv ORA_OCI_UCBPKG "pkg1;pkg2;pkg3;pkg4;pkg5"
```

#### Example 14-3 Specifying the pkgNInit() and PkgNEnvCallback() Functions

```
pkgNInit(void *metaCtx, void *libCtx, ub4 argfmt, sword argc, void **argv)
{
  return OCISharedLibInit(metaCtx, libCtx, argfmt, argc, argv, pkgNEnvCallback);
}
```

#### Example 14-4 Using pkgNEnvCallback() to Register Entry, Replacement, and Exit Callbacks

```
pkgNEnvCallback(OCIEnv *env, ub4 mode, size_t xtramemsz,
                                void *usrmemp, OCIUcb *ucbDesc)
{
  OCIHandleAlloc((void *)env, (void **)&errh, OCI_HTYPE_ERROR, (size_t) 0,
        (void **)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;
}
```

#### Example 14-5 Registering User Callbacks with the NULL ucbDesc

```
  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);
```

#### Example 14-6 Using the OCIStmtPrepare() Call to Call the Callbacks in 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()
```

**See Also:**

OCIStmtPrepare2()

## 14.2 OCI Callbacks from External Procedures

Provides additional reference information about using OCI callbacks from external procedures.

There are several OCI functions that you can use as callbacks from external procedures.

**See Also:**

- OCI Cartridge Functions for a list of functions you can use as callbacks from external procedures
- *Oracle Database Development Guide* for information about writing C subroutines that can be called from PL/SQL code, including a list of which OCI calls you can use and some example code

# 15

# Performance Topics

This chapter describes topics about OCI performance features.

You can use OCI to access Oracle TimesTen In-Memory Database and Oracle TimesTen Application-Tier Database Cache.

This chapter includes the following topics:

- Statement Caching in OCI
- Implicit Fetching of ROWIDs
- OCI Support for Implicit Results
- Client Result Cache
- Client Statement Cache Auto-Tuning

## 15.1 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. You can use statement caching with connection pooling and with session pooling, and improve performance and scalability. You can use it without session pooling as well. OCI calls that implement statement caching are:

- `OCIStmtPrepare2()`
- `OCIStmtRelease()`

This section includes the following topics:

- Statement Caching Without Session Pooling in OCI
- Statement Caching with Session Pooling in OCI
- Rules for Statement Caching in OCI
- Bind and Define Optimization in Statement Caching
- OCI Statement Caching Code Example

**See Also:**

- OCIStmtPrepare2()
- OCIStmtRelease()

## 15.1.1 Statement Caching Without Session Pooling in OCI

To perform statement caching without session pooling, users perform the usual OCI steps to log on.

The call to obtain a session has a mode that specifies whether statement caching is enabled for the session. Initially the statement cache is empty. Developers try to find a statement in the cache using the statement text. If the statement exists, the API returns a previously prepared statement handle; otherwise, it returns a newly prepared statement handle.

The application developer can perform binds and defines and then simply execute and fetch the statement before returning the statement to the cache. If the statement handle is not found in the cache, the developer must set different attributes on the handle in addition to the other steps.

`OCIStmtPrepare2()` takes a mode that determines if the developer wants a prepared statement handle or a null statement handle if the statement is not found in the cache.

The pseudocode looks like this:

```
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, ...);
OCIStmtFetch2(svchp, ...);
OCIStmtRelease(stmthp, ...);
...
```

**See Also:**

OCIStmtPrepare2()

## 15.1.2 Statement Caching with Session Pooling in OCI

For statement caching with session pooling, 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. It is set on the `OCI_HTYPE_SPOOL` handle. The statement cache size for a particular session in the pool can be overridden at any time by using `OCI_ATTR_STMTCACHESIZE` on that session. The value of `OCI_ATTR_SPOOL_STMTCACHESIZE` can be changed at any time. You can use this attribute 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 is reflected on individual sessions in the pool, when they are provided to a user. Tagged sessions are an exception to this behavior. This is explained later in this section.

**Note:**

You can change the attributes after acquiring a session. However, once an attribute is changed, it will remain set on the underlying physical session. This value will not be reset back implicitly while releasing the session back to the session pool. Hence, it is the developer's responsibility to maintain the state of the sessions before releasing the session using `OCIStmtRelease()`.

Enabling or disabling of statement caching is allowed on individual pooled sessions as it is on nonpooled 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 defaults 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 is 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 does not hold true if a tagged session is asked for and retrieved. In this case, the size of the statement cache is not changed. Consequently, it is not turned on or off. Moreover, if the user specifies mode `OCI_SESSGET_STMTCACHE` in the `OCISessionGet()` call, this is ignored if the session is tagged. In our earlier example, if Session A was tagged, then it is returned as is to the user.

**See Also:**

- OCISessionGet()
- OCILogon2()

## 15.1.3 Rules for Statement Caching in OCI

If you are using statement caching, follow these rules.

Here are some rules to follow for statement caching in OCI:

- 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 raises 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 frees all buffers associated with the old session when the statement handle is migrated.

- You are required to keep one service context per session. Any statement handle obtained using `OCIStmtPrepare2()` with a certain service context should be subsequently used only in conjunction with the same service context, and never with a different service context.
- A call to `OCIStmtPrepare2()`, even if the session does not have a statement cache, also allocates the statement handle. Therefore, applications using only `OCIStmtPrepare2()` must not call `OCIHandleAlloc()` for the statement handle.
- A call to `OCIStmtPrepare2()` must be followed by a call to `OCIStmtRelease()` after the user is done with the statement handle. If statement caching is used, this releases the statement to the cache. If statement caching is not used, the statement is 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) that 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 nonpooled. If `OCISessionBegin()` is called without the 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()` returns a non-statement cache-enabled session. You can use `OCI_ATTR_STMTCACHESIZE` to turn the caching on. Similarly, you can use the same attribute to turn off statement caching by setting the cache size to zero.
- You can tag a statement at the release time so that the next time you can request a statement of the same tag. The tag is 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 differ in their tags, or if one is untagged and the other is not.

**See Also:**

- OCIStmtPrepare2()
- OCIHandleAlloc()
- OCIStmtRelease()
- OCIHandleFree()
- OCISessionBegin()
- OCISessionGet()
- Statement Functions
- Service Context Handle Attributes
- Session Pool Handle Attributes

## 15.1.4 Bind and Define Optimization in Statement Caching

To avoid repeated bind and define operations on statements in the cache by the application, the application can register an opaque context with a statement taken from the statement cache and register a callback function with the service context.

The application data such as bind and define buffers can be enclosed in the opaque context. This context is registered with the statement the first time it is taken from the cache. When a statement is taken from the cache the second time and onwards, the application can reuse the bind and define buffers, that it had registered with that statement. It is still the application's responsibility to manage the bind and defines. It can reuse both the bind and define data and the buffers, or it can change only the data and reuse the buffers, or it can free and reallocate the buffers if the current size is not enough. In the last case, it must rebind and redefine. To clean up the memory allocated by the application toward these bind and define buffers, the callback function is called during aging out of the statement or purging of the whole cache as part of session closure. The callback is called for every statement being purged. The application frees the memory and does any other cleanup required, inside the callback function. Example 15-1 shows the pseudocode.

**Example 15-1 Optimizing Bind and Define Operations on Statements in the Cache**

```
Get the statement using OCIStmtPrepare2(...)

Get the opaque context from the statement if it exists

If opaque context does not exist

{

  Allocate fetch buffers, do the OCIBindByPos, OCIDefineByPos, and so forth

  Enclose the buffer addresses inside a context and set the context and
  callback function on the statement

}
Execute/Fetch using the statement, and process the data in the fetch buffers.
```

```
OCIStmtRelease() that statement

Next OCIStmtPrepare2()

OCIAttrGet() opaque application context from statement handle

Execute/Fetch using the statement and process the data in the fetch buffers.

OCIStmtRelease()

. . .

void callback_fn (context, statement, mode)

{

   /* mode= OCI_CBK_STMTCACHE_STMTPURGE means this was called when statement was
      aging out of the statement cache or if the session is ended */

  <free the buffers in the context.>

}
```

**See Also:**

- OCI_ATTR_STMTCACHE_CBKCTX
- OCI_ATTR_STMTCACHE_CBK

### 15.1.5 OCI Statement Caching Code Example

Indicates where to find a working example of statement caching.

See `cdemostc.c` in directory `demo` for a working example of statement caching.

## 15.2 Implicit Fetching of ROWIDs

This section describes the following topics:

- About Implicit Fetching of ROWIDs
- Example of Implicit Fetching of ROWIDs

### 15.2.1 About Implicit Fetching of ROWIDs

`ROWID` is a globally unique identifier for a row in a database. It is created at the time the row is inserted into the table, and destroyed when it is removed.

`ROWID` values have several important uses. They are unique identifiers for rows in a table. They are the fastest way to access a single row and can show how the rows in the table are stored.

Implicit fetching of `ROWID`s in `SELECT ... FOR UPDATE` statements means that the `ROWID` is retrieved at the client side, even if it is not one of the columns named in the select statement. The `position` parameter of `OCIDefineByPos()` is set to zero (0). These host variables can be specified for retrieving the `ROWID` pseudocolumn values:

- `SQLT_CHR` (`VARCHAR2`)
- `SQLT_VCS` (`VARCHAR`)
- `SQLT_STR` (`NULL`-terminated string)
- `SQLT_LVC` (`LONG VARCHAR`)
- `SQLT_AFC` (`CHAR`)
- `SQLT_AVC` (`CHARZ`)
- `SQLT_VST` (OCI String)
- `SQLT_RDD` (`ROWID` descriptor)

The `SELECT ... FOR UPDATE` statement identifies the rows that are to be updated and then locks each row in the result set. This is useful when you want to base an update on the existing values in a row. In that case, you must ensure that another user does not change the row.

When you specify character buffers for storing the values of the `ROWID`s (for example, if getting it in `SQLT_STR` format), allocate enough memory for storing `ROWID`s. Remember the differences between the `ROWID` data type and the `UROWID` data type. The `ROWID` data type can only store physical `ROWID`s, but `UROWID` can store logical `ROWID`s (identifiers for the rows of index-organized tables) as well. The maximum internal length for the `ROWID` type is 10 bytes; it is 3950 bytes for the `UROWID` data type.

Dynamic define is equivalent to calling `OCIDefineByPos()` or `OCIDefineByPos2()` with `mode` set as `OCI_DYNAMIC_FETCH`. Dynamic defines enable you to set up additional attributes for a particular define handle. It specifies a callback function that is invoked at runtime to get a pointer to the buffer into which the fetched data or a piece of it is to be retrieved.

The attribute `OCI_ATTR_FETCH_ROWID` must be set on the statement handle before you can use implicit fetching of `ROWID`s, in this way:

```
OCIAttrSet(stmthp, OCI_HTYPE_STMT, 0, 0 , OCI_ATTR_FETCH_ROWID, errhp);
```

Dynamic define is not compatible with implicit fetching of `ROWID`s. In normal scenarios this mode allows the application to provide buffers for a column, for each row; that is, a callback is invoked every time a column value is fetched.

This feature, using `OCIDefineByPos()` or `OCIDefineByPos2()` for position 0, is for fetching an array of data simultaneously into the user buffers and getting their respective `ROWID`s at the same time. It allows for fetching of `ROWID`s with `SELECT....FOR UPDATE` statements even when `ROWID` is not one of the columns in the `SELECT` query. When fetching the data one by one into the user buffers, you can use the existing attribute `OCI_ATTR_ROWID`.

If you use this feature to fetch the `ROWID`s, the attribute `OCI_ATTR_ROWID` on the statement handle cannot be used simultaneously to get the `ROWID`s. You can only use one of them at a time for a particular statement handle.

**See Also:**

- OCI_ATTR_FETCH_ROWID
- OCIDefineByPos() or OCIDefineByPos2()

## 15.2.2 Example of Implicit Fetching of ROWIDs

Shows an example of implicit fetching of ROWIDs.

Use the fragment of a C program in Example 15-2 to build upon.

**Example 15-2 Implicit Fetching of ROWIDs**

```
#include <oci.h>

int main()
{
 ...
 text *mySql = (text *) "SELECT emp_name FROM emp FOR UPDATE";
 text rowid[100][15] = {0};
 text empName[100][15] = {0};
 ...

 /* Set up the environment, error handle, etc. */
 ...

 /* Prepare the statement -  select ... for update. */


  if (OCIStmtPrepare (select_p, errhp,
                      mySql, strlen(mySql), OCI_NTV_SYNTAX, OCI_DEFAULT))
  {
    printf ("Prepare failed \n");
    return (OCI_ERROR);
  }

 /* Set attribute for implicit fetching of ROWIDs on the statement handle. */
 if (OCIAttrSet(select_p, OCI_HTYPE_STMT, 0, 0, OCI_ATTR_FETCH_ROWID, errhp))
 {
   printf ("Unable to set the attribute - OCI_ATTR_FETCH_ROWID \n");
   return OCI_ERROR;
 }
  /*
   * Define the positions: 0 for getting ROWIDs and other positions
   * to fetch other columns.
   * Also, get the define conversion done implicitly by fetching
   * the ROWIDs in the string format.
   */

  if (OCIDefineByPos ( select_p,
                       &defnp0,
                       errhp,
                       0,
                       rowid[0],
                       15,
                       SQLT_STR,
                       (void *) ind,
                       (void *) 0,
                       (void *) 0,
                       OCI_DEFAULT) ||
       OCIDefineByPos(select_p,
                       &defnp1,
                       errhp,
                       1,
                       empName[0],
                       15,
                       SQLT_STR,
                       (void *) 0,
                       (void *) 0,
                       (void *) 0,
                       OCI_DEFAULT)
                       )
  {
    printf ("Failed to define\n");
    return (OCI_ERROR);
  }


  /* Execute the statement. */

 if (errr = OCIStmtExecute(svchp,
                            select_p,
                            errhp,
                            (ub4) 5,
                            (ub4) 0,
                            (OCISnapshot *) NULL,
                            (OCISnapshot *) NULL,
                            (ub4) OCI_DEFAULT))
  {
    if (errr != OCI_NO_DATA)
       return errr;
  }

  printf ("Column 0  \t Column 1\n");
  printf ("_________ \t ________\n");

  for (i =0 ;i<5 i++)
   {
     printf("%s \t %s \n", rowid[i], empName[i]);
   }

 return OCI_SUCCESS;
}
```

## 15.3 OCI Support for Implicit Results

Beginning with Oracle Database 12*c* Release 1 (12.1) , PL/SQL can return results (cursors) implicitly from stored procedures and anonymous PL/SQL blocks. `OCIStmtGetNextResult()` is provided to retrieve and process the implicit results.

PL/SQL provides a subprogram `RETURN_RESULT` in the `DBMS_SQL` package to return the result of an executed statement as shown in Example 15-3. In the current release, only `SELECT` query result-sets can be implicitly returned by a PL/SQL procedure block. `OCIStmtGetNextResult()` returns an OCI statement handle on which the usual OCI define and fetch calls are done to retrieve the rows.

Example 15-4 shows a PL/SQL stored procedure to implicitly return result-sets (cursors) to the client.

Example 15-5 shows the same approach using an anonymous PL/SQL block sent by the client. This example shows how applications can use the implicit results feature to implement batching of SQL statements from an OCI application. An OCI application can dynamically form a PL/SQL anonymous block to execute multiple and variable `SELECT` statements and return the corresponding cursors using `DBMS_SQL.RETURN_RESULT`.

Example 15-6 lists an OCI program showing how to use the `OCIStmtGetNextResult()` call to retrieve and process the implicit results returned by a PL/SQL stored procedure (see Example 15-4) or an anonymous PL/SQL block (see Example 15-5).

`OCIStmtGetNextResult()` can be called iteratively by the application to retrieve each implicit result from an executed PL/SQL statement. Applications retrieve each result-set sequentially but can fetch rows from any result-set independently. The top-level OCI statement handle tracks all the associated result-set statement handles. Freeing or releasing the top-level OCI statement handle automatically closes and frees all the implicit result-sets.

The attribute `OCI_ATTR_IMPLICIT_RESULT_COUNT` is provided on the OCI statement handle to determine the number of implicit results available.

The `rtype` parameter of `OCIStmtGetNextResult()` returns the type of the result. In this release only the type: `OCI_RESULT_TYPE_SELECT` is supported. The describe metadata of the returned result set can be accessed similar to any `SELECT` ResultSet.

> **Note:**
>
> The following OCI code can be used in external procedures too, to fetch from the implicit results. In that case, `OCI_PREP2_IMPL_RESULTS_CLIENT` should be passed as the mode to the `OCIStmtPrepare2()` call.

**Example 15-3 DBMS_SQL RETURN_RESULT Subprogram**

```
procedure return_result(rc        in out sys_refcursor,
                        to_client in boolean default true);

procedure return_result(rc        in out integer,
                        to_client in boolean default true);
```

**Example 15-4 A PL/SQL Stored Procedure to Implicitly Return Result-Sets (Cursors) to the Client**

```
CREATE PROCEDURE foo AS
  c1 sys_refcursor;
  c2 sys_refcursor;
begin
  open c1 for select * from emp;
  dbms_sql.return_result(c1); --return to client
  -- open 1 more cursor
  open c2 for select * from dept;
  dbms_sql.return_result (c2); --return to client
end;
```

### Example 15-5 An Anonymous PL/SQL Block to Implicitly Return Result-Sets (Cursors) to the Client

```
declare
  c1 sys_refcursor;
  c2 sys_refcursor;
begin
  open c1 for select * from emp;
  dbms_sql.return_result (c1); --return to client
  -- open 1 more cursor
  open c2 for select * from dept;
  dbms_sql.return_result (c2); --return to client
end;
```

### Example 15-6 Using OCIStmtGetNextResult() to Retrieve and Process the Implicit Results Returned by Either a PL/SQL Stored Procedure or Anonymous Block

```
  OCIStmt *stmthp;
  ub4      rsetcnt;
  void    *result;
  ub4      rtype;
  char    *sql = "begin foo; end;";

  /* Prepare and execute the PL/SQL procedure. */
  OCIStmtPrepare2(svchp, &stmthp, errhp, (oratext *)sql, strlen(sql),
                 NULL, 0, OCI_NTV_SYNTAX, OCI_DEFAULT);
  OCIStmtExecute(svchp, stmthp, errhp, 1, 0,
                 (const OCISnapshot *)0,
                 (OCISnapshot *)0, OCI_DEFAULT);

  /* Now check if any implicit results are available. */
  OCIAttrGet((void *)stmthp, OCI_HTYPE_STMT, &rsetcnt, 0,
             OCI_ATTR_IMPLICIT_RESULT_COUNT, errhp);

  /* Loop and retrieve the implicit result-sets.
   * ResultSets are returned in the same order as in the PL/SQL
   * procedure/block.
   */
  while (OCIStmtGetNextResult(stmthp, errhp, &result, &rtype,
                              OCI_DEFAULT) == OCI_SUCCESS)
  {
    /* Check the type of implicit ResultSet, currently
     * only supported type is OCI_RESULT_TYPE_SELECT
     */
    if (rtype == OCI_RESULT_TYPE_SELECT)
    {
      OCIStmt *rsethp = (OCIStmt *)result;

      /* Perform normal OCI actions to define and fetch rows. */
    }
    else
      printf("unknown result type %d\n", rtype);

    /* The result set handle should not be freed by the user. */
  }

  OCIStmtRelease(stmthp, errhp, NULL, 0, OCI_DEFAULT); /* Releases the statement
handle. */
```

**See Also:**

- OCIStmtGetNextResult()
- Statement Handle Attributes for more information about the attribute `OCI_ATTR_IMPLICIT_RESULT_COUNT`.
- OCIStmtPrepare2()

# 15.4 Client Result Cache

OCI applications can use client memory to take advantage of the OCI result cache to improve response times of repeated queries.

**See Also:**

*Oracle Database Development Guide* for complete information about using the OCI client result cache

# 15.5 Client Statement Cache Auto-Tuning

Describes topics about client statement cache auto-tuning.

This section describes the following topics:

- About Auto-Tuning Client Statement Cache
- Benefit of Auto-Tuning Client Statement Cache
- Client Statement Cache Auto-Tuning Parameters
- Usage Examples of Client Statement Cache Auto Tuning
- Enabling and Disabling OCI Client Auto-Tuning
- Usage Guidelines for Auto-Tuning Client Statement Cache

## 15.5.1 About Auto-Tuning Client Statement Cache

Auto-tuning optimizes OCI client session features of mid-tier applications to gain higher performance without the need to reprogram your OCI application.

Auto tuning operations like increasing or decreasing cache memory happens implicitly during `OCIStmtPrepare2()` and `OCIStmtRelease()` calls on a periodic basis. Calling `OCIAttrGet()` with `OCI_ATTR_STMTCACHESIZE` on the service handle gives the current cache size being used, if you need to check.

It is possible for the coded OCI client statement cache size setting to be sub optimal. This can happen, for example, with changing the workload causing a different working set of SQL statements. If the size is too low, it will cause excess network activity and more parses at the server. If the size is too high, there will be excess memory used. It can be difficult for the client side application to always keep this cache size optimal.

Auto-tuning automatically reconfigures the OCI statement cache size on a periodic basis. Auto-tuning is achieved by providing a deployment time setting that provides an option to reconfigure OCI statement caching to resolve this potential performance issue.

These settings are provided as connect string based deployment settings in the client `oraaccess.xml` file that overrides manual settings to the user configuration of OCI features.

> **See Also:**
>
> - OCIStmtPrepare2()
> - OCIStmtRelease()

## 15.5.2 Benefit of Auto-Tuning Client Statement Cache

The more specific benefit of auto-tuning client statement cache is to transparently detect, monitor, and adjust the statement cache size to improve performance or decrease memory usage.

Developers and DBAs can expect to see the following benefits of using auto-tuning for their OCI client applications:

- Reduced time and effort in diagnosing and fixing performance problems with each part of their system, such as statement caching
- Minimized manual modifications needed to configurations of this OCI feature to improve performance. Usually, this manual correction requires applications to restart more than once with different configuration parameters, thus further reducing the high availability of the client
- One solution that can be used by all OCI applications to improve performance right out-of-the-box without having to make any application changes
- OCI applications are freed from making custom implementations (that can be error prone) to auto-tune their OCI application to optimize performance and memory usage. Here auto-tuning is limited to internal automatic tuning of OCI Client-side statement cache size only.

## 15.5.3 Client Statement Cache Auto-Tuning Parameters

The following connection specific parameters in `oraccess.xml` can be set per configuration alias or across all connect strings using default connection specific parameters.

Values specified in the client `oraaccess.xml` configuration file override programmatic settings.

This section includes the following topics:

- <statement_cache>
- <auto_tune>
- Comparison of the Connection Specific Auto-Tuning Parameters

**See Also:**

About Specifying Defaults for Connection Parameters for more information about setting per configuration alias or across all connect strings using default connection specific parameters

### 15.5.3.1 <statement_cache>

This parameter is optional and sets the limit for the statement caching tunable component.

```
<statement_cache>
  <size>100</size>
</statement_cache>
```

The limit is the maximum number of statements that can be cached per session. If auto-tuning is enabled or not, this setting in `oraaccess.xml` overrides the programmatic setting of OCI statement cache size.

If auto-tuning is enabled, this setting will be the upper bound on statement cache size while its being dynamically tuned.

If the session is not using statement caching APIs as in `OCIStmtPrepare2()` and `OCIStmtRelease()`, this setting is ignored.

Default values are as follows:

- If auto-tuning is enabled, statement caching is dynamically tuned and the initial statement cache size is set to 100 statements.
- If auto-tuning is disabled, this setting serves as the deployment setting of statement caching size, overriding any programmatic setting.

**See Also:**

- OCIStmtPrepare2()
- OCIStmtRelease()
- Enabling and Disabling OCI Client Auto-Tuning

### 15.5.3.2 <auto_tune>

This section specifies auto tune parameters.

If the OCI session is not using statement caching APIs as in `OCIStmtPrepare2()` or `OCIStmtRelease()`, auto tuning parameters are ignored for that session. It is possible in a process that some sessions or connections can have auto-tuning enabled and some disabled.

This section includes the following topics:

- <enable>true</enable>

- <ram_threshold>
- <memory_target>

**See Also:**

- OCIStmtPrepare2()
- OCIStmtRelease()

### 15.5.3.2.1 <enable>true</enable>

This parameter turns auto tuning on or off.

The default is auto tuning off `(FALSE)` or disabled.

```
<auto_tune>
  <enable>true</enable>
</auto_tune>
```

Auto-tuning is enabled along with internal default settings.

**See Also:**

<statement_cache> for more information about auto-tuning is enabled along with internal default settings

### 15.5.3.2.2 <ram_threshold>

This parameter is optional.

```
<auto_tune>
   <enable>true</enable>
   <ram_threshold>0.1</ram_threshold>
</auto_tune>
```

The default value is 0.01%. It is specified as percentage of installed RAM. This specifies the total memory available across the auto tuning sessions in a process sharing this setting. This setting can be specified per process or per connect string alias.

Note that if specified per connect string alias, the total auto tuning memory used by a client process can add up.

Therefore, it may be preferable to specify auto tuning limits in the `<default_parameters>` section of `oraaccess.xml` file. This way you have a common pool of memory for all sessions in a client process.

A smaller limit uses less RAM for auto tuning, but minimizes the chance other programs running on the system do not degrade in performance.

This parameter must be specified within the `<auto_tune></auto_tune>` deployment setting.

**See Also:**

About oraaccess.xml

### 15.5.3.2.3 <memory_target>

This parameter is optional.

```
<auto_tune>
  <enable>true</enable>
  <memory_target>40M</memory_target>
</auto_tune>
```

Specified in bytes. Default is undefined. It specifies the total memory available across the auto tuning sessions in a process sharing this setting. This setting can be specified per process or per connect string alias.

Note that if specified per connect string alias, the total auto tuning memory used by a client process can add up.

Therefore, it may be preferable to specify auto tuning limits in the `<default_parameters>` section of `oraaccess.xml` file. This way you have a common pool of memory for all sessions in a client process.

This parameter must be specified within the `<auto_tune></auto_tune>` deployment setting.

Using this parameter ensures the use of a consistent memory limit for auto tuning irrespective of installed RAM on that system.

If not specified, the auto tuning memory limit is based on the `<ram_threshold>` parameter setting.

If both `<ram_threshold>` and `<memory_target>` parameters are specified, the effective limit is the minimum of the two parameters.

**See Also:**

About oraaccess.xml

## 15.5.3.3 Comparison of the Connection Specific Auto-Tuning Parameters

Lists and describes a comparison of all auto-tuning parameters.

Table 15-1 shows a comparison of the connection specific auto-tuning parameters.

**Table 15-1 Comparison of Some Connection Specific Auto-Tuning Parameters**

| Parameter | Setting and Semantics | For Auto-Tuning or Deployment Setting |
|---|---|---|
| `<statement_cache>` | Optional setting.<br>Per session cache size. | If auto-tuning is enabled (see "Enabling and Disabling OCI Client Auto-Tuning"), this is the upper bound of each sessions statement cache size while its tuned by auto tuning.<br>Or else it refers to the deployment setting for statement caching. |
| `<auto_tune>` | Optional setting.<br>Specify this parameter to use auto-tuning. Applies to all connections using this connect string or all connections if null connect string is specified. | Only auto-tuning related |
| `<ram_threshold>0.1</ram_threshold>` | Optional setting.<br>Converts the percentage setting to a memory value based on installed RAM on that client or mid-tier system.<br>This is the upper limit of memory used for auto tuning within a client process.<br>For installed RAM of 8GB, not specifying this parameter gives 800 KB of memory among the sessions.<br>Note each connection can potentially have its own setting of auto tuning parameters so these values can add up for the whole process based on configuration settings. It is preferable to use this parameter hence in the `<default_parameters>` section of the `oraaccess.xml` file. See "File (oraaccess.xml) Properties" for a description of the syntax. | Only auto-tuning related. If auto-tuning is disabled, this parameter setting is ignored. This parameter must be specified within the `<auto_tune></auto_tune>` deployment setting. |

**Table 15-1 (Cont.) Comparison of Some Connection Specific Auto-Tuning Parameters**

| Parameter | Setting and Semantics | For Auto-Tuning or Deployment Setting |
|---|---|---|
| `<memory_target>1048576</memory_target>` | Optional setting.<br>This is the upper limit of memory used for auto tuning within a client process.<br>Note each connection can potentially have its own setting of auto tuning parameters so these values can add up for the whole process based on configuration settings. It is preferable to use this parameter hence in the `<default_parameters>` section of the `oraaccess.xml` file.<br>See "File (oraaccess.xml) Properties" for a description of the syntax.<br>Value is in bytes. 1,048,576 bytes is 1 MB. | Only auto-tuning related. If auto-tuning is disabled, this parameter setting is ignored. This parameter must be specified within the `<auto_tune></auto_tune>` deployment setting. |

## 15.5.4 Usage Examples of Client Statement Cache Auto Tuning

The following are some usage examples showing use and interaction of client statement cache auto-tuning parameters that are also connection specific parameters.

```
<statement_cache>
  <size>100</size>
</statement_cache>
```

The programmatic statement cache size will be replaced by this setting. Auto-tuning is disabled and cache is managed per LRU. In this case, the application developer believes the OCI application statement prefetching programmatic settings do not need to be overridden.

```
<auto_tune>
  <enable>true</enable>
</auto_tune>
```

Auto-tuning is enabled along with internal default settings.

```
<statement_cache>
  <size>100</size>
</statement_cache>
<auto_tune>
  <enable>true</enable>
  <memory_target>40M</memory_target>
</auto_tune>
```

This statement caching deployment setting of 100 will replace the programmatic statement cache size and because auto-tuning is enabled, statement caching will be auto-tuned. The memory target setting is in effect because auto-tuning is enabled.

Auto tuning will always try to limit total statement cache memory used around a memory target. If a memory target is not specified, it is based on the percentage of total installed RAM.

In this case, the memory limit is the specified memory target.

> **See Also:**
>
> <statement_cache> for more information about auto-tuning is enabled along with internal default settings

## 15.5.5 Enabling and Disabling OCI Client Auto-Tuning

Describes conditions that enable and disable OCI client auto-tuning.

The following conditions enable and disable OCI client auto-tuning:

- Auto-tuning is enabled when the client `oraaccess.xml` `<auto_tune>` section is added with enable specified as true, `<enable>true</enable>`
- Auto-tuning is disabled by default or when enable is set to false, <enable>false</enable> in `oraaccess.xml` under the `<auto_tune>` section.

## 15.5.6 Usage Guidelines for Auto-Tuning Client Statement Cache

Describes guidelines to use when setting the auto-tuning parameters.

The following are some guidelines to use when setting the auto-tuning parameters:

- When either client response, memory allocation, or client CPU is high and you want to gain performance without rebuilding the OCI application, you can use `<auto_tune>` settings or deployment `<statement_cache>` settings. Auto tuning may also decrease the network bytes transferred between client and server.
- When AWR or ADDM reports lots of parses and you cannot or you may prefer not to programmatically modify the statement cache size, you can specify auto-tuning for statement cache or use the deployment statement cache setting `<statement_cache>`.

# 16

# Database Startup and Shutdown

This chapter describes topics about OCI database startup and shutdown.

This section describes the following topics:

- About OCI Database Startup and Shutdown
- Examples of Startup and Shutdown in OCI

## 16.1 About OCI Database Startup and Shutdown

The OCI functions `OCIDBStartup()` and `OCIDBShutdown()` provide the minimal interface needed to start 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 start the instance. A call to `OCIDBStartup()` starts 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, 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` is signaled if an attempt is made to start or shut down the database when it is connected to a shared server through a dispatcher.

The `OCIAdmin` administration handle C data type 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()
> - OCIDBShutdown()
> - Administration Handle Attributes
> - *Oracle Database Administrator’s Guide*

## 16.2 Examples of Startup and Shutdown in OCI

To perform 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`) is used. In the latter case, pass `(OCIAdmin *)0`. A call to `OCIDBStartup()` starts one instance on the server.

Example 16-1 shows sample code that uses a client-side parameter file (`pfile`) that is set in the administration handle and performs a database startup operation.

To perform 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`, use the following procedure:

1. Call `OCIDBShutdown()` in `OCI_DEFAULT`, `OCI_DBSHUTDOWN_TRANSACTIONAL`, `OCI_DBSHUTDOWN_TRANSACTIONAL_LOCAL`, or `OCI_DBSHUTDOWN_IMMEDIATE` mode to prohibit further connections.
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 16-1 Calling OCIDBStartup() to Perform a Database Startup Operation**

```
...

/*  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((void *) envhp, (void **) &admhp,
         (ub4) OCI_HTYPE_ADMIN, (size_t) 0, (void **) 0));

/* Set attribute pfile in the admin handle
(do not do this if you want to use the spfile) */
checkerr (errhp, OCIAttrSet( (void *) admhp, (ub4) OCI_HTYPE_ADMIN,
          (void *) pfile, (ub4) strlen(pfile),
          (ub4) OCI_ATTR_ADMIN_PFILE, (OCIError *) errhp));

/* Start up in NOMOUNT mode */
  checkerr(errhp, OCIDBStartup(svchp, errhp, admhp, OCI_DEFAULT, 0));
   checkerr(errhp, OCIHandleFree((void *) 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);
...
```

### Example 16-2    Calling OCIDBShutdown() in OCI_DBSHUTDOWN_FINAL Mode

```
/*  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));
...
```

**Example 16-3    Calling OCIDBShutdown() in 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_DBSHUTDOWN_ABORT));

/* End the sysdba session */
checkerr(errhp, OCISessionEnd(svchp, errhp, usrhp, (ub4)OCI_DEFAULT));
...
```

Example 16-2 shows sample code that uses a client-side parameter file (`pfile`) that is set in the administration handle that performs an orderly database shutdown operation.

Example 16-3 shows a shutdown example that uses `OCI_DBSHUTDOWN_ABORT` mode.

**See Also:**

- OCIAttrSet()
- OCIDBStartup()
- OCIDBShutdown()

# 17

# Support for Pluggable Databases

The multitenant architecture enables an Oracle database to contain a portable collection of schemas, schema objects, and nonschema objects that appear to an Oracle client as a separate database.

A multitenant container database (CDB) is an Oracle database that includes one or more pluggable databases (PDBs).

OCI clients can connect to a PDB using a service whose pluggable database property has been set to the relevant PDB.

In general, OCI calls behave the same way whether connected to a pluggable database or a normal database. OCI calls and features that require special consideration with a CDB are described in the sections that follow:

- Enhancements on OCI API Calls with Multitenant Container Databases (CDB) in General
- OCI Enhancements for ALTER SESSION SET CONTAINER
- Restrictions on OCI API Calls with Multitenant Container Databases (CDB) in General
- Restrictions on OCI Calls with ALTER SESSION SET CONTAINER
- Restrictions on OCI Calls with ALTER SESSION SWITCH CONTAINER SWITCH SERVICE

> **See Also:**
>
> *Oracle Database Administrator’s Guide* for more information about PDBs and for more details about configuring the services to connect to various PDBs

## 17.1 Enhancements on OCI API Calls with Multitenant Container Databases (CDB) in General

These enhancements are the result of restrictions that are now removed for Oracle Database 12*c* Release 2 (12.2) and later.

The following enhancements are the result of restrictions that are now removed for Oracle Database 12*c* Release 2 (12.2) and later.

- Client result cache works with connections to Pluggable Databases.

## 17.2 OCI Enhancements for ALTER SESSION SET CONTAINER

Some OCI restrictions about using the `ALTER SESSION SET CONTAINER` statement are removed.

The client must be Oracle Database Release 12*c* Release 2 (12.2) to perform these operations. Lower version clients are returned an error if any of the following operations is attempted.

- The `ALTER SESSION SET CONTAINER` statement supports applications that use `TIMESTAMP WITH TIMEZONE` or `TIMESTAMP WITH LOCAL TIMEZONE` data types in OCI. The semantics of these types are preserved when the application switches between pluggable databases having different database time zone settings or different database time zone file version settings.
- The `ALTER SESSION SET CONTAINER` statement supports the use of this command to switch an OCI connection between any two pluggable databases whose character sets are different.

## 17.3 Restrictions on OCI API Calls with Multitenant Container Databases (CDB) in General

Descriptions general restrictions.

- An attempt to logon in `OCI_PRELIM_AUTH` mode when connected to any container other than `CDB$ROOT` will result in an `ORA-24542` error.
- An attempt to issue `OCIDBStartup()` when connected to any container other than `CDB$ROOT` results in an `ORA-24543` error.
- An attempt to issue `OCIDBShutdown()` when connected to any container other than `CDB$ROOT` results in an `ORA-24543` error. When `OCIDBShutdown()` is issued connected to `CDB$ROOT`, it brings down the whole instance.
- OCI Continuous Query Notification (CQN) is not supported with CDB.
- OCI applications linked against a client library older than release 12.1 or higher and connecting to a pluggable database will not be able to utilize Fast Application Notification (FAN) High Availability (HA) functionality when connected as a normal (non-common) user. As a workaround, such applications should connect as a common user. This restriction does not exist for release 12.1 or higher OCI clients.

> **See Also:**
>
> - OCIDBStartup()
> - OCIDBShutdown()

# 17.4 Restrictions on OCI Calls with ALTER SESSION SET CONTAINER

Describes specific restrictions.

The `ALTER SESSION SET CONTAINER` statement can be used to switch an OCI connection from one pluggable database to another. However, applications that use the `ALTER SESSION SET CONTAINER` statement to switch between pluggable databases need to ensure that their usage is consistent with the OCI restrictions described as follows.

- The `ALTER SESSION SET CONTAINER` statement is disallowed for OCI migratable sessions (such as sessions created with `OCI_MIGRATE` mode during logon) and the combination results in an `ORA-65135` error.
- The `ALTER SESSION SET CONTAINER` statement is not supported with OCI connection pool (which is the old OCI connection pool API) and the combination results in an `ORA-65135` error.
- The `ALTER SESSION SET CONTAINER` statement is not supported in conjunction with OCI session switching (wherein multiple OCI user handles share the same OCI server handle).
- If the client initially connects to a container with a `EXTENDED MAX_STRING_SIZE` setting, and then within the same session switches to a container (using an `ALTER SESSION SET CONTAINER` statement) with an `STANDARD MAX_STRING_SIZE` setting, then a subsequent `OCIStmtExecute()` call will result in an `ORA-14697` error if an attempt is made to use any bind variables of size greater than 4000 bytes.
- An attempt to fetch from an OCI statement handle using `OCIStmtFetch()` or `OCIStmtFetch2()` in the context of a different container than the one in which it was executed will result in an `ORA-65108` error.
- OCI client result cache is disabled if an `ALTER SESSION SET CONTAINER` statement is done in OCI.
- Fast Application Notification (FAN) and Runtime Connection Load Balancing notifications are not supported for applications that switch connections between pluggable databases using an `ALTER SESSION SET CONTAINER` statement.
- The `ALTER SESSION SET CONTAINER` statement sets the current transaction, if any, to read only and any attempt to perform any of the OCI transaction calls (`OCITransStart()`, `OCITransDetach()`, `OCITransCommit()`, `OCITransRollback()`, `OCITransPrepare()`, `OCITransMultiPrepare()`, `OCITransForget()`) will return an error in the new container. In order to issue any of these calls, you need to switch back to the original container.
- If an `OCISubscriptionUnRegister()` call is attempted in the context of an incorrect container (different from the container on which the corresponding `OCISubscriptionRegister()` call was done), then an `ORA-24950` is returned.
- A `OCIDescribeAny()` call with `OCI_PTYPE_DATABASE` describes the database to which the connection is connected. After an `ALTER SESSION SET CONTAINER` statement is done, if the application wants to see the current database description, the `OCIDescribeAny()` call will need to be reissued.

- Calls to any OCI Any Data, collection, or object functions that are used to manipulate an object from a different container are not supported.
- An `OCIObjectFlush()` call is supported only in the container where the object instance was created with an `OCIObjectNew()` call.
- Oracle recommends that `OCIObjectFlush()` be called prior to switching containers with an `ALTER SESSION SET CONTAINER` statement. Note that an `OCIObjectFlush()` call will start a transaction if one is not already started.
- An `OCIObjectFlush()` call done after switching containers may return an error if a transaction was already started earlier on another container by the same session (either as a result of explicit DMLs or as a result of an `OCIObjectFlush()`call).
- An `OCIObjectFlush()` call only flushes objects dirtied in the context of the container in which the `OCIObjectFlush()` call is issued.
- Various session attributes may change on an `ALTER SESSION SET CONTAINER` statement. If an application caches these attributes, their settings may no longer be the same after an `ALTER SESSION SET CONTAINER` statement. Examples of attributes that can be obtained with an `OCIAttrGet()` call and which can change on an `ALTER SESSION SET CONTAINER` statement include the following:
  - OCI_ATTR_CURRENT_SCHEMA
  - OCI_ATTR_INITIAL_CLIENT_ROLES
  - OCI_ATTR_EDITION
  - OCI_ATTR_MAX_OPEN_CURSORS

**See Also:**

- *Oracle Database SQL Language Reference* for more information about `MAX_STRING_SIZE`
- OCIStmtExecute()
- OCIStmtFetch() or OCIStmtFetch2()
- OCITransStart(), OCITransDetach(), OCITransCommit(), OCITransRollback(), OCITransPrepare(), OCITransMultiPrepare() , OCITransForget()
- OCISubscriptionUnRegister() and OCISubscriptionRegister()
- OCIDescribeAny()
- OCIObjectFlush()
- OCIObjectNew()
- OCIAttrGet()

## 17.5 Restrictions on OCI Calls with ALTER SESSION SWITCH CONTAINER SWITCH SERVICE

Describes a restriction on OCI calls with the `ALTER SESSION SWITCH CONTAINER SWITCH SERVICE` statement.

Beginning with Oracle Database 12*c* Release 2 (12.2), the `ALTER SESSION SWITCH CONTAINER SWITCH SERVICE` statement is added, which may result in a change of service attributes if the new service is configured differently than when connected to the original service. However, OCI will not alter its processing based on the new settings with this statement; instead, it will continue to use the original settings as when connected to the original service. For example, FAN on/off, TAF on/off settings, and so forth will remain the same as when first connected to the original service. This is considered the desired behavior after the switch as the normal use case for the `ALTER SESSION SWITCH CONTAINER SWITCH SERVICE` statement is in a multitenant environment with the same application being in use after the switch.

## 18

# OCI Interface for Using Shards

This chapter describes the OCI interface for using Oracle Sharding.

Shards are locations in a set of databases where each database stores some part of the data. The part of the data stored at each database is represented by a set of *chunks*, where each chunk is associated with a certain range of the data.

To make requests that read from or write to a chunk, your application must be routed to the appropriate database (shard) that stores that chunk during the connection initiation step. This routing is accomplished by using a data key. The data key enables routing to the specific chunk by specifying its sharding key or to a group of chunks by specifying its super sharding key. In order to get a connection to the correct shard containing the chunk you wish to operate on, you must specify a key in your application before getting a connection to a sharded Oracle database for either standalone connections or connections obtained from an OCI Session pool. For an OCI Session pool, you must specify a data key before you check out connections from the pool.

For OCI Session pools and stand alone connections, the steps to form sharding keys and shard group keys and get a session with an underlying connection include:

1. Allocate the sharding key descriptor by calling `OCIDescriptorAlloc()` and specifying the descriptor type parameter as `OCI_DTYPE_SHARDING_KEY` to form the sharding key.
   a. Add all of the columns of the sharding key by calling `OCIShardingKeyColumnAdd()` as many times as is needed to form the complete sharding key.
   b. Call `OCIAttrSet()` and specify the `OCI_ATTR_SHARDING_KEY` attribute to set the sharding key on the authentication handle.
2. Allocate the shard group key descriptor by calling `OCIDescriptorAlloc()` and specifying the descriptor type parameter as `OCI_DTYPE_SHARDING_KEY` to form the shard group key.
   a. Add all of the group columns of the sharding key by calling `OCIShardingKeyColumnAdd()` as many times as is needed to form the complete shard group key.
   b. Call `OCIAttrSet()` and specify the `OCI_ATTR_SUPER_SHARDING_KEY` attribute to set the shard group key on the authentication handle.
3. Call `OCISessionGet()` using the initialized authentication handle from the previous step containing the sharding key and shard group key information to get the database connection to the shard and chunk specified by the sharding key and group of chunks as specified by the shard group key.

For custom pools, the steps to form sharding keys and shard group keys and check out a connection from the pool include:

1. If there are no existing connections in your custom pool, go directly to Step 7. Otherwise, do the following steps.

2. Allocate the sharding key descriptor by calling `OCIDescriptorAlloc()` and specifying the descriptor type parameter as `OCI_DTYPE_SHARDING_KEY` to form the sharding key. Add all of the columns of the sharding key by calling `OCIShardingKeyColumnAdd()` as many times as is needed to form the complete sharding key.
3. Allocate the shard group key descriptor by calling `OCIDescriptorAlloc()` and specifying the descriptor type parameter as `OCI_DTYPE_SHARDING_KEY` to form the shard group key. Add all of the group columns of the sharding key by calling `OCIShardingKeyColumnAdd()` as many times as is needed to form the complete shard group key.
4. Call `OCIShardInstancesGet()` with the sharding key, super sharding key descriptors and the connect string to return the instance name or names that contain the desired chunk for the specified sharding key descriptor and super sharding key descriptor.
5. Examine each connection in the custom pool to see if it points to one of the instances whose name was returned by `OCIShardInstancesGet()`. To check the instance that a connection in your custom pool points to, you can get the instance name using `OCI_ATTR_INSTNAME` on the service context handle (`OCISvcCtx *`).
6. If you find a suitable connection in your custom pool that points to one of the desired instances, then call `OCIAttrSet()` to associate the sharding key and super sharding key with that connection. Now the connection is ready for executing your application's OCI calls on the desired shard. You can skip Step 7. If you do not find a suitable connection in your custom pool that points to the desired instance, continue to Step 7.
7. If there is no matching connection found, create a new connection with the sharding key, super sharding key, and connect string and call `OCISessionGet()` and specify the `OCI_SESSGET_CUSTOM_POOL` mode to explicitly cache the shard topology that contains the shard to chunking mapping information. Now you have a connection to the desired shard. Now the connection is ready for executing your application's OCI calls on the desired shard.

Providing the sharding key or a super sharding key to OCI enables it to get a connection to the desired shard. As previously noted, these keys must be specified before getting a standalone connection to the database, or before checking out a connection from the OCI Session pool so that an appropriate connection to the desired shard can be returned.

In the case of custom pools, when the pool is empty, the custom pool implementer can use the steps as described above to first create standalone connections to the desired shards by providing the sharding key and super sharding key (with `OCI_SESSGET_CUSTOM_POOL` mode) to populate the custom pool, and secondly, for subsequent requests for connections to specific shards, use the `OCIShardInstancesGet()` call as described in conjunction with `OCIAttrGet()` of `OCI_ATTR_INSTNAME` to determine if an existing connection to the desired shard already exists in the custom pool, and if it does, then the connection can be reused.

The section describes the OCI interfaces that:

- Create a sharding key and a super sharding key using OCI data types.
- Create a connection specifying a sharding key and a super sharding key.
- Specify the sharding key and super sharding key to the connection request from an OCI session pool.

- Use custom connection pooling to get the shard name for a given connection and to get the shard name and the chunk name, given a sharding key and super sharding key.

### About Specifying a Sharding Key and Super Sharding Keys for a Standalone Connection

Use the `OCISessionGet()` call for creating the connection. This call takes an authentication handle, `authp`, as input, on which various properties are set, including the two attributes to support creating connections to specific shards: `OCI_ATTR_SHARDING_KEY` for the sharding key and `OCI_ATTR_SUPER_SHARDING_KEY` for the super sharding key.

### About Creating a Sharding Key and Super Sharding Key

Use the OCI descriptor type `OCIShardingKey` for forming the sharding key and the super sharding key. This descriptor wraps the key value (for a single part key) or multiple values (for a compounded key).

Use the following `OCIShardingKeyColumnAdd()` call to add all of the columns of the key to form the complete key.

```
OCIShardingKeyColumnAdd(OCIShardingKey *shardingKey,
                     OCIError    *errhp,
                     void        *col,
                     ub4          colLen,
                     ub2          colType,
                     ub4          mode)
```

You should perform this call as many times as there are columns in the compound key (or just once for a simple sharding key) in the order in which the key is defined in the database. The `columnType` parameter indicates the data type of the column.

The following table shows the supported OCI data type values for the `columnType` parameter and its corresponding C data type.

| OCI Data Type | C Data Type |
|---|---|
| SQLT_NUM | ub1* |
| SQLT_CHR | OraText* |
| SQLT_DATE | ub1* |
| SQLT_TIMESTAMP | OCIDateTime* |
| SQLT_RAW | ub1* |
| SQLT_VNU | ub1* |
| SQLY_INT | int* |

The character key values are assumed to be in the client character set (specified by `NLS_LANG` or `OCIEnvNLSCreate()` calls) .

After forming the sharding key and the super sharding key using the`OCIShardingKeyColumnAdd()` call, the keys can be set on the authentication handle using the sharding key attribute `OCI_ATTR_SHARDING_KEY` and the super sharding key attribute `OCI_ATTR_SUPER_SHARDING_KEY` as follows:

```
OCIAttrSet(authp,
           OCI_HTYPE_AUTHINFO,
           shardKey,
           sizeof(shardKey),
           OCI_ATTR_SHARDING_KEY,
           errhp);


OCIAttrSet(authp,
           OCI_HTYPE_AUTHINFO,
           shardGroupKey,
           sizeof(shardGroupKey),
           OCI_ATTR_SUPER_SHARDING_KEY,
           errhp);
```

When you use this `authp` parameter in an `OCISessionGet()` call, it will ensure that you create a connection to the shard containing the data corresponding to the sharding key and super sharding key values that are set.

### About Getting the Actual Sharding Key and Super Sharding Key Values

If you want to know the Base64 representation of the sharding key and super sharding key for diagnostic purposes, you can use the attribute `OCI_ATTR_SHARDING_KEY_B64`, on the `OCIShardingKey` descriptor. The `OCIAttrGet()` call takes the `OCIShardingKey` descriptor as input and returns the text value in base64 format of the sharding key and super sharding key.

```
OCIAttrGet((dvoid *) OCIShardingKey,
   (ub4)             OCI_DTYPE_SHARDING_KEY,
   (dvoid *)         &sekyVale,
   (ub4*)            &skeyValueLen,
                     OCI_ATTR_SHARDING_KEY_B64,
   (OCIError *)      errhp);
```

In addition, you can use the `OCIShardingKeyReset()` call shown as follows for your application to reset and reuse the allocated descriptor for creating a new sharding key and super sharding key.

```
sword OCIShardingKeyReset(OCIShardingKey *shardKey,
                          OCIError       *errhp,
                          ub4             mode);
```

This section includes the following topics:

- About Specifying a Sharding Key and Super Sharding Key for Getting a Connection from an OCI Session Pool
- About Specifying a Sharding Key and Super Sharding Key for Getting a Connection from a Custom Pool

**See Also:**

- OCIShardingKeyColumnAdd()
- OCIShardingKeyReset()

## 18.1 About Specifying a Sharding Key and Super Sharding Key for Getting a Connection from an OCI Session Pool

This topic describes how to specify a sharding key and super sharding key to get a connection from an OCI Session Pool.

By default, the `OCISessionGet()` call creates a new connection. You can also use this call to get an existing connection from an OCI Session pool. When you use the `OCI_ATTR_SHARDING_KEY` and `OCI_ATTR_SUPER_SHARDING_KEY` attribute settings, you can get a connection to the desired shard from an OCI Session pool. In an OCI Session pool, sessions in the pool can represent a variety of shards that are each authenticated by the database credentials passed in by the `OCISessionGet()` call using the initialized authentication handle `authp`.

The following example shows how to get a connection to the desired database shard from an OCI Session Pool created with the homogeneous and statement caching modes specified. Even though this example uses a homogeneous pool, you are not restricted to that type of pool.

```
OCIShardingKey   *shardKey, *shardGroupKey;
  /* Error handling is omitted for brevity. */
  /* Create a homogeneous session pool. */
  checkerr(&status, errhp,
           OCISessionPoolCreate(envhp, errhp,
                 spoolhp,                                     /* session
pool handle */
                (OraText **) poolName, poolNameLenp,          /* returned
poolname, length */
                (const OraText *) connstr, strlen(connstr),   /* connect
string */
                 min, max, increment,                         /* pool size
constraints */
                (OraText *) "hr", strlen((char *) "hr"),      /* username
*/
                (OraText *) apppassword,                      /* password
*/
                 strlen((char *) apppassword),
                 OCI_SPC_HOMOGENEOUS|OCI_SPC_STMTCACHE));     /* modes */

/* Allocate the sharding key and super sharding key descriptors. */
OCIDescriptorAlloc(envhp,(dvoid **)&shardKey,
               OCI_DTYPE_SHARDING_KEY, 0,(dvoid **)0)))
text *name = “KK”;
text *gname = “GOLD”;
int  empid = 150;
```

```
/* Add all the columns of the key to form the final sharding key. */
OCIShardingKeyColumnAdd(shardKey,(ub1*)&empid, sizeof(empid),
                               SQLT_INT, errhp, OCI_DEFAULT);
OCIShardingKeyColumnAdd(shardKey, name, strlen(name),
                               SQLT_CHAR, errhp, OCI_DEFAULT));

OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
           shardKey, sizeof(shardKey),
           OCI_ATTR_SHARDING_KEY, errhp);

/* Setting a shard group key. */
/* Create a shard group key, in the same way as for a sharding key. */
OCIDescriptorAlloc(envhp,(dvoid **)&shardGroupKey,
                   OCI_DTYPE_SHARDING_KEY, 0, (dvoid **)0));

/* Add the column of the key to form the final super sharding key. */
OCIShardingKeyColumnAdd(shardGroupKey, gname, strlen(gname),
                     SQLT_CHAR, errhp, OCI_DEFAULT));

OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
           shardGroupKey, sizeof(shardGroupKey),
           OCI_ATTR_SUPER_SHARDING_KEY, errhp));

/* Get the database connection from the OCI Session Pool. */
checkerr(&status,
         errhp, OCISessionGet(envhp, errhp,
                &svchp,                             /* returned database
connection */
                authp,                              /* initialized
authentication handle */
               (OraText *) poolName, poolNameLen,    /* connect string */
                NULL, 0, NULL, NULL, NULL,           /* session tagging
parameters: optional */
                OCI_DEFAULT));                      /* modes */
```

#### Chunk Migrations and OCISessionGet() in Sharding

During chunk migrations, when a chunk migrates from one shard instance to another, OCI Session Pool can ensure that `OCISessionGet()` returns a connection to an instance having a writable chunk by implicitly doing retrials during chunk migration. This requires that a couple of properties be set:

- Setting `READONLY_CHUNK_OK` to `FALSE` in the connect string.
- Setting pool handle attributes: `OCI_ATTR_SPOOL_GETMODE` and `OCI_ATTR_SPOOL_WAIT_TIMEOUT` to `OCI_SPOOL_ATTRVAL_TIMEDWAIT` and a suitable time out value in milliseconds. If the pool is unable to get a connection to a writable instance within the time out period, `OCISessionGet()` returns an `ORA-24495` error.

If the application can use read only chunks, it can set `READONLY_CHUNK_OK=true` in the connect string. In that case a connection available to the instance with chunks marked read-only also may be dispensed. If the application attempts any database write operation on such a connection, it gets in return suitable errors.

# 18.2 About Specifying a Sharding Key and Super Sharding Key for Getting a Connection from a Custom Pool

This topic describes features that support specifying a sharding key and super sharding key to get a connection from a custom pool.

This section describes the following features that support applications specifying a sharding key and super sharding key for getting a connection from a custom pool:

- `OCISessionGet()` mode `OCI_SESSGET_CUSTOM_POOL` — Used to explicitly cache the shard topology every time OCI connects to a new shard instance.
- `OCIShardInstancesGet()` — Returns instance names for a given sharding key descriptor, super sharding key descriptor, and connection string.

**OCI_ATTR_INSTNAME Attribute**

If your OCI client application uses custom connection pooling, you must be able to return connections to specific shards. To do this, you must know the shard name to which shard a connection has been made and the sharding key and super sharding key to shard name mapping to enable a look up for a matching connection.

This is possible when you use the attribute `OCI_ATTR_INSTNAME` on the service context (`svchp`) for this purpose. This attribute returns the instance name for a given connection. The instance name is unique for the shard instance pointed to by a given connection. Every shard instance has a unique name. The following code sample shows how this attribute is used to get the instance name from a given service context `svchp`.

```
OraText shardName[OCI_INSTNAME_MAXLEN];
ub4 shardNameLen;
   OCIAttrGet(svchp,
   OCI_HTYPE_SVCCTX,
   shardName,
   (ub4 *) &shardNameLen,
   OCI_ATTR_INSTNAME,
   errhp);
```

**OCISessionGet() mode OCI_SESSGET_CUSTOM_POOL**

OCI clients that do custom pooling must use the `OCISessionGet()` mode `OCI_SESSGET_CUSTOM_POOL` to explicitly cache the shard topology that contains the shard to chunking mapping information every time OCI connects to a new shard not previously visited. OCI clients that do not use custom pooling do not need to use this mode as this caching is done implicitly, for example, when you use OCI Session Pools.

**OCIShardInstancesGet()**

`OCIShardInstancesGet()` returns instance names for a given sharding key descriptor and super sharding key descriptor. This method has the following signature:

```
sword OCIShardInstancesGet(
           void            **shTopoCtx,
        OCIError         *errhp,
        const OraText    *connstr,
        ub4               constrl,
        OCIShardingKey   *shardingKey,
        OCIShardingKey   *superShardingKey,
        OCIShardinst   ***shardinsts,
        ub4               numShardInsts,
        ub4               mode);
```

This call looks up the shard topology cache on the client.

If any connections were made with `OCI_SESSGET_CUSTOM_POOL` mode, then OCI maintains the shard topology cache locally.

The returned value may be `NULL` if the shard topology cache does not yet have the mapping either because no connections were made with `OCI_SESSGET_CUSTOM_POOL` or the connections that were made so far with `OCI_SESSGET_CUSTOM_POOL` did not connect to a shard that contains the chunk for the sharding key and super sharding key requested. In either case, the custom pool should create a new connection explicitly using `OCISessionGet()` with the `OCI_ATTR_SHARDING_KEY` and `OCI_ATTR_SUPER_SHARDING_KEY` attributes set as applicable with `OCI_SESSGET_CUSTOM_POOL` mode enabled. Doing this adds information about the new shard to the OCI shard topology cache. Subsequent `OCIShardInstancesGet()` calls will look for key ranges belonging to chunks located in these shards and return these shard instance names.

Note that a sharding key and super sharding key may refer to multiple shards as a result of the chunk being replicated. When the custom pool has a connection to the desired shard, the custom pool should ensure that the `OCI_ATTR_SHARDING_KEY` and `OCI_ATTR_SUPER_SHARDING_KEY` properties are set on the connection before dispensing. Setting these properties ensures that the chunk usage is tracked on the database side and to determine if there have been any chunk splits occurring on the database.

**See Also:**

OCIShardInstancesGet() for an example that uses custom pooling

# 19

# OCI Object-Relational Programming

This chapter introduces the OCI facility for working with objects in an Oracle database. It also discusses the object navigational function calls of OCI.

This chapter contains these topics:

- OCI Object Overview
- About Working with Objects in OCI
- About Developing an OCI Object Application
- About Type Inheritance
- About Type Evolution

## 19.1 OCI Object Overview

OCI allows applications to access any of the data types found in Oracle Database, 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

> **Note:**
>
> Beginning with Oracle Database 12c Release 2 (12.2), there is a change in the linear snapshot size from 24 bytes to 34 bytes that requires the reconstruction of the collection image form. When an older client or server accesses a Release 12.2 or higher database involving any access to collections, the collection image form that contains the linear snapshot must be converted when sending or receiving to older clients or server. This conversion results in the performance degradation. Oracle recommends that you use a 12.2 version of the client or server to avoid this conversion.

To take full advantage of Oracle Database object capabilities, most applications must 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 the OCI 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

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.

Applications interacting with Oracle Database objects need a way to represent those objects in a host language format. Oracle Database 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. OCI includes a set of *data type mapping and manipulation functions* that enable an application to manipulate these data types, and thus manipulate the attributes of objects.

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.

**See Also:**

- About Developing an OCI Object Application for more detail about OCI navigational calls
- Describing Schema Metadata for a discussion of `OCIDescribeAny()`
- Object-Relational Data Types in OCI for a more detailed discussion about functions
- OCIDescribeAny()

## 19.2 About 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 the OCI object functions. An object can also be directly manipulated as a value instance (without using its object reference).

This section includes the following topics:

- Basic Object Program Structure
- Persistent Objects, Transient Objects, and Values

## 19.2.1 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.

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 must include C struct representations of database objects in a header file. These structs can be created by the programmer, or, more easily, they can be generated by the Object Type Translator (OTT).
2. Allocate necessary handles, and establish a connection to a server.
3. Prepare a SQL statement for execution. This is a local (client-side) step, which may include binding placeholders and defining output variables. In an object-relational application, this SQL statement should return a reference (`REF`) to an object.

   > **Note:**
   >
   > It is also possible to fetch an entire object, rather than just a reference (`REF`). If you select a referenceable object, rather than pinning it, you get that object *by value*. You can also select a nonreferenceable object.
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 can do some or all of the following:

   - Manipulate the attributes of the object and mark it as *dirty* (modified)
   - 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 reexecute prepared statements again.

These steps are discussed in more detail in the remainder of this chapter.

**See Also:**

- Overview of OCI Program Programming
- Fetching Embedded Objects describes fetching the entire object
- OCI Programming Basics for information about using OCI to connect to a server, process SQL statements, and allocate handles and the description of the OCI relational functions in Oracle Database Access C API
- About Representing Objects in C Applications for information about OTT and Using the Object Type Translator with OCI

## 19.2.2 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 they are referenceable by way of an object identifier.

**Note:**

The terms *object* and *instance* are used interchangeably in this manual.

This section includes the following topics:

- Persistent Objects
- Transient Objects
- Values

**See Also:**

*Oracle Database Object-Relational Developer's Guide* for more information about objects

### 19.2.2.1 Persistent Objects

A persistent object is an object that 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 that 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 an object table, and each instance 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 objects 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 an object table. They are embedded within other structures. Examples of embedded objects are objects that are attributes of another object, or instances that exist in an object column of a database table. Embedded instances do not have object identifiers, and OCI applications cannot get `REF`s to embedded instances.

  Embedded objects may also be referred to as *nonreferenceable 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.

Example 19-1 and Example 19-2 show SQL examples that demonstrate the difference between these two types of persistent objects.

Objects that 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.

Objects that 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*.

**Example 19-1    SQL Definition of Standalone Objects**

```
CREATE TYPE person_t AS OBJECT
   (name      varchar2(30),
    age       number(3));
CREATE TABLE person_tab OF person_t;
```

**Example 19-2    SQL Definition of Embedded Objects**

```
CREATE TABLE department
   (deptno     number,
    deptname   varchar2(30),
    manager    person_t);
```

## 19.2.2.2 Transient Objects

A transient object is a temporary instance whose life does not exceed that of the application, and that 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:**

- About Creating Objects for more information about using `OCIObjectNew()`
- OCIObjectNew()

### 19.2.2.3 Values

A value is referred to as being either a scalar value or as an embedded or nonreferenceable object.

In the context of this manual, a *value* refers to either:

- A scalar value that 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 nonreferenceable object.

The context should make it clear which meaning is intended.

**Note:**

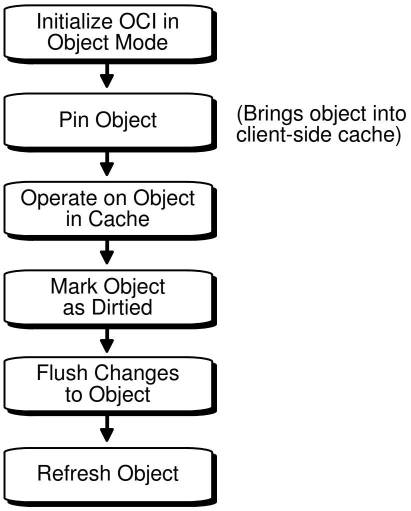
It is possible to select a referenceable object into the object cache, rather than pinning it, in which case you fetch the object *by value* instead of fetching its `REF`.

## 19.3 About Developing an OCI Object Application

This section discusses the steps involved in developing a basic OCI object application.

Figure 19-1 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 19-1 Basic Object Operational Flow**


Initialize OCI in Object Mode
Pin Object
(Brings object into client-side cache)
Operate on Object in Cache
Mark Object as Dirtied
Flush Changes to Object
Refresh Object

This section includes the following topics:

- About Representing Objects in C Applications
- About Initializing the Environment and the Object Cache
- About Making Database Connections
- Retrieving an Object Reference from the Server
- Pinning an Object
- Manipulating Object Attributes
- About Marking Objects and Flushing Changes
- Fetching Embedded Objects
- Object Meta-Attributes
- Complex Object Retrieval
- COR Prefetching
- OCI Versus SQL Access to Objects
- Pin Count and Unpinning
- NULL Indicator Structure
- About Creating Objects
- About Freeing and Copying Objects
- Object Reference and Type Reference
- Create Objects Based on Object Views and Object Tables with Primary-Key-Based OIDs
- Error Handling in Object Applications

**See Also:**

Basic Object Program Structure

## 19.3.1 About 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 database 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 must 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.

Oracle Database provides a utility called the Object Type Translator (OTT), which generates C struct representations of database object types for you. For example, suppose that you have a type in your database declared as follows:

```
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 functions are mentioned later in this chapter.

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:**
>
> - Object Cache and Memory Management for application programmers who want to use object representations other than the default structs generated by the object cache
> - NULL Indicator Structure
> - Using the Object Type Translator with OCI for more information about OTT
> - Object-Relational Data Types in OCI for more detail about the subset of OCI functions that manipulate data of these variable types used in the struct declarations employed by the OCI object calls

### 19.3.2 About Initializing the Environment and the Object Cache

If your OCI application is going to access and manipulate 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 is working with objects.

This notification has the following important effects:

- It establishes the *object runtime 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 `OCIEnvCreate()` or `OCIEnvNlsCreate()` is not set to `OCI_OBJECT`, any attempt to use an object-related function results 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:**

- Object Advanced Topics in OCI for a detailed explanation of the object cache
- OCIEnvCreate()
- OCIEnvNlsCreate()

## 19.3.3 About 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. When you use these calls, no additional considerations must be made because this application is accessing objects.

Only one object cache is 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.

**See Also:**

OCI Programming Steps for more information about properly initializing the OCI environment

## 19.3.4 Retrieving an Object Reference from the Server

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 `REF`s to one or more objects.

**Note:**

It is also possible for a SQL statement to fetch embedded objects, rather than `REF`s, from a database.

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, when given a particular employee number that 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 as follows in the same way that it would handle any relational SQL statement:

1. Prepare an application request, using `OCIStmtPrepare2()`.
2. Bind the host input variable using one or more appropriate bind calls.
3. Declare and prepare an output variable to receive the employee object reference. Here you would use an employee object reference:

   ```
   OCIRef  *emp1_ref = (OCIRef *) 0; /* reference to an employee object */
   ```

   When you define the output variable, set the *dty* data type parameter for the define call to SQLT_REF, the data type constant for `REF`.
4. Execute the statement with `OCUStmtExecute()`.
5. Fetch the resulting `REF` into `emp1_ref`, using `OCIStmtFetch2()`.

At this point, you could use the object reference to access and manipulate an object or objects from the database.

**See Also:**

- Fetching Embedded Objects for more information
- OCI Programming Basics
- OCI Programming Steps for general information about preparing and executing SQL statements
- Advanced Bind Operations in OCI and Advanced Define Operations in OCI for specific information about binding and defining `REF` variables
- The demonstration programs included with your Oracle installation for a code example showing `REF` retrieval and pinning. For additional information, see OCI Demonstration Programs.
- OCIStmtPrepare2()
- About Representing Objects in C Applications about using an employee object reference declaration mentioned in Step 3
- OCIStmtExecute()
- OCIStmtFetch2()

## 19.3.5 Pinning an Object

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.

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*. 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 Complex Object Retrieval.

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.

Example 19-3 shows sample code that illustrates a pin operation for the employee reference your application retrieved in the previous section, Retrieving an Object Reference from the Server.

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.
- `OCI_DURATION_TRANS` is the pin duration.
- `OCI_LOCK_X` is the lock option.
- `emp1` is an out parameter that 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.

This section includes the following topic: Array Pin.

**Example 19-3 Pinning an Object**

```
if (OCIObjectPin(env, err, emp1_ref, (OCIComplexObject *) 0,
     OCI_PIN_ANY,
     OCI_DURATION_TRANS,
     OCI_LOCK_X,  &emp1) != OCI_SUCCESS)
     process_error(err);
```

**See Also:**

- Simple Object Navigation for an example of navigation from one instance to another
- OCIObjectPin()
- About Pinning an Object Copy for more information about the pin option `OCI_PIN_ANY`
- Object Duration for more information about the pin duration `OCI_DURATION_TRANS`
- About Locking Objects for Update for more information about the lock option `OCI_LOCK_X`

### 19.3.5.1 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 to fetch objects of different types from different tables in one network round-trip.

**See Also:**

OCIObjectArrayPin()

## 19.3.6 Manipulating Object Attributes

Once an object has been pinned, an OCI application can modify its attributes.

OCI provides a set of functions for working with data types of object type structs, known as the OCI *data type 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. For changes made by the application to reach the database, those changes must be flushed or committed to the server.

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, you must access the employee's hire date and check whether it is more or less than 180 days before 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 in Example 19-4 demonstrates this process.

Note that the data type mapping and manipulation functions work with a specific set of data types; you must convert other types, like `int`, to the appropriate OCI types before using them in calculations.

Example 19-4 points out how values must be converted to OCI data types (for example, `OCIDate`, `OCINumber`) before being passed as parameters to the OCI data type mapping and manipulation functions.

**Example 19-4 Manipulating Object Attributes in OCI**

```
/* 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, (void  *)&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);
```

**See Also:**

- About Marking Objects and Flushing Changes for more information about flushing changes or committing them to the server
- Object-Relational Data Types in OCI for more information about the OCI data types and the data type mapping and manipulation functions

## 19.3.7 About Marking Objects and Flushing Changes

The application must take specific steps to ensure that changes to objects by marking them and then flushing them are written in the database.

In Example 19-4, 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 ensure 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 OCI traverses the dirty list maintained by the cache and flushes the dirty objects to the server.
- Call `OCICacheFlush()` to commit a transaction. Doing so also traverses the dirty list and flushes the dirty 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:**

- OCI Support for Transactions for more information about `OCITransCommit()`
- About Creating Objects for information about transient and persistent objects
- Object Meta-Attributes for information about seeing and checking object meta-attributes, such as *dirty*
- OCIObjectMarkUpdate()
- OCIObjectFlush()
- OCICacheFlush()
- OCICacheFlush()

## 19.3.8 Fetching Embedded Objects

An application must fetch embedded object instances.

If your application must 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 Retrieving an Object Reference from the Server. Embedded instances do not have object identifiers, so it is not possible to get a `REF` to them; they cannot serve as the basis for object navigation. Many situations exist, however, in which an application must 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 data type 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 OCI Programming Basics:

- Prepare an application request, using `OCIStmtPrepare2()`.
- Bind the input variable using one or more appropriate bind calls.
- 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 About Representing Objects in C Applications.

  ```
  addr1      *address; /* variable of the address struct type */
  ```

  When you define the output variable, set the `dty` data type parameter for the define call to SQLT_NTY, the data type constant for named data types.
- Execute the statement with `OCIStmtExecute()`.
- Fetch the resulting instance into `addr1`, using `OCIStmtFetch2()`.

Following this operation, you can access the attributes of the instance, as described in Manipulating Object Attributes, 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:**

- OCI Programming Steps for more information about preparing and executing SQL statements
- OCIStmtPrepare2()
- OCIStmtExecute()
- OCIStmtFetch2()

## 19.3.9 Object Meta-Attributes

An object's *meta-attributes* serve as flags that 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 it has been flushed to the server. Object meta-attributes 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 subdivided 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.

This section includes the following topics:

- Persistent Object Meta-Attributes
- Additional Attribute Functions
- Transient Object Meta-Attributes

### 19.3.9.1 Persistent Object Meta-Attributes

Lists and describes the meta-attributes for *standalone* persistent objects.

Table 19-1 shows the meta-attributes for *standalone* persistent objects.

**Table 19-1 Meta-Attributes of Persistent Objects**

| Meta-Attributes | Meaning |
|---|---|
| existent | Does the object exist? |
| nullity | Null information of the instance |
| locked | Has the object been locked? |
| dirty | Has the object been marked as *dirtied?* |
| pinned | Is the object pinned? |
| allocation duration | See Object Duration. |
| pin duration | See Object Duration. |

> **Note:**
>
> Embedded persistent objects only have the *nullity* and *allocation duration* attributes, which are transient.

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 void         *obj,
                             OCIObjectPropId    propertyId,
                             void               *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.

**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 communicates 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 the size of the 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 communicates 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 the size of the 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`

**OCI_OBJECTPROP_LOCK**
This returns the lock status of the object. The possible lock status is indicated 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`. 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`, 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.

> **See Also:**
>
> - OCIObjectGetProperty()
> - Object Duration for more information about durations
> - OCIObjectIsLocked()

### 19.3.9.2 Additional Attribute Functions

Lists and describes additional attribute functions known as set and check functions.

OCI also provides functions that allow an application to set or check some of these attributes directly or indirectly, as shown in Table 19-2.

**Table 19-2 Set and Check Functions**

| Meta-Attribute | Set with | Check with |
|---|---|---|
| nullity | <none> | OCIObjectGetInd() |
| existence | <none> | OCIObjectExists() |
| locked | OCIObjectLock() | OCIObjectIsLocked() |
| dirty | OCIObjectMarkUpdate() | OCIObjectIsDirty() |

### 19.3.9.3 Transient Object Meta-Attributes

Lists and describes transient object meta-attributes.

Transient objects have no persistent attributes. Table 19-3 shows the following transient attributes.

**Table 19-3 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. |
| pin duration | See Object Duration. |

## 19.3.10 Complex Object Retrieval

A complex object includes its root object and its set of logically related objects each of which are prefetched based on a given depth level.

In Example 19-3 and Example 19-4, 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 these 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 with objects can be improved with *complex object retrieval (COR)*. This is a prefetching mechanism in which an application specifies the 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 must 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, but it improves performance. Its use is optional.

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:

- A `REF` to the desired root object.
- 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 preceding purchase order object, the application must specify the following:

- The `REF` to the root purchase order object
- One or more pairs of type and depth information for `cust`, `related_orders`, or `line_items`

An application fetching a purchase order may 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 that 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, as follows:

- `REF(PO object)`
- `{(customer, 1)}`

For the application 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.

- `REF(PO object)`
- `{(customer, UB4MAXVAL), (purchase_order, UB4MAXVAL)}`

(In this example, `UB4MAXVAL` specifies that all objects of the specified type reachable through references from the root object should be prefetched.)

For an application to fetch a PO and all the associated line items, it would specify:

- `REF(PO object)`
- `{(line_item, 1)}`

The application can also fetch all objects reachable from the root object by way of `REF`s (transitive closure) by setting 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. Although, doing so results in many extraneous fetches, quite simple to specify and requires only one server round-trip.

This section includes the following topics:

- About Prefetching Objects
- About Implementing Complex Object Retrieval in OCI

## 19.3.10.1 About 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 been prefetched and are in the object cache.

Consider 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 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 incurs a network round-trip when those objects are accessed later.

The `READ` or `SELECT` privilege is needed for all prefetched objects. Objects in the complex object for which the application does not have `READ` or `SELECT` privilege are not prefetched.

### 19.3.10.2 About 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 specify 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 `REF`s 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_TYPE_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 that 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:**

- Handles
- OCI Descriptors
- OCIObjectPin()
- OCIAttrSet()
- OCIDescriptorAlloc()
- OCIParamSet()
- OCIAttrGet()
- OCIObjectPin()

## 19.3.11 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 graphs of objects rooted at a given root object. The traversal stops when all required objects have been prefetched, or when the total size of all the prefetched objects exceeds the *prefetch limit*.

This section includes the following topics:

- COR Interface
- Example of COR

### 19.3.11.1 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,
                     void               **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,
                          void              **obj_array,
                          ub4               *pos );
```

Note the following points when using COR:

- A null COR handle argument defaults to pinning just the root object.
- A COR handle with the 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.
- The lock options apply only to the root object.

> **Note:**
>
> 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()`).

> **See Also:**
>
> - OCIObjectPin()
> - OCIObjectArrayPin()
> - OCIObjectArrayPin()

### 19.3.11.2 Example of COR

Shows how an application program can be modified to use complex object retrieval.

Example 19-5 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.

**Example 19-5 Using Complex Object Retrieval in OCI**

```
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((void  *) envhp, (void  **) &corhp, (ub4)
                OCI_HTYPE_COMPLEXOBJECT, 0, (void  **)0);

/* get COR descriptor for type line_item */
OCIDescriptorAlloc((void  *) envhp, (void  **) &cordp, (ub4)
                OCI_DTYPE_COMPLEXOBJECTCOMP, 0, (void  **) 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( (void  *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP,
            (void  *) litdo, (ub4) sizeof(void  *), (ub4)
            OCI_ATTR_COMPLEXOBJECTCOMP_TYPE, (OCIError *) errhp);
level = 1;

/* set depth level for line_item_varray in COR descriptor */
OCIAttrSet( (void  *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP,
            (void  *) &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, (void  **)&po);

/* free COR descriptor and COR handle */
OCIDescriptorFree((void  *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP);
OCIHandleFree((void  *) 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, (void  **)&liref, (void  **)0, &eoc);
while (!eoc)        /* not end of collection */
{
/* pin line item */
   OCIObjectPin(envhp, errhp, *liref, (void  *)0, OCI_PIN_RECENT,
       OCI_DURATION_SESSION,
       OCI_LOCK_NONE, (void  **)&li));
  display_line_item(li);

/* get next line item */
OCIIterNext(envhp, errhp, itr, (void  **)&liref, (void  **)0, &eoc);
}
```

## 19.3.12 OCI Versus SQL Access to Objects

If an application must manipulate a graph of objects (interrelated 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, requiring multiple network round-trips. Using the complex object retrieval capability provided by 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 wants 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 execution of corresponding `INSERT` and `DELETE` statements would also be required. 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`, and `DELETE` statements.

Using the `OCICacheFlush()` function, the application can flush all modifications (insertion, deletion, and update of objects) in a single operation. OCI does all the bookkeeping, thereby requiring less coding in the application. For manipulating a graph of objects OCI is not only efficient, but also provides an easy-to-use interface.

Consider a different case in which the application must fetch an object when given its `REF`. In 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 is fetched from the server only once for the transaction, and repeated pins on the same `REF` return a pointer to the pinned object in the cache. In the SQL interface, each execution of the `SELECT DEREF...` statement would result in fetching the object from the server. This would result in multiple round-trips to the server and multiple copies of the same object.

Finally, consider the case in which the application must fetch a nonreferenceable object, as in the following example:

```
CREATE TABLE department
(
deptno number,
deptname varchar2(30),
manager employee_t
);
```

The `employee_t` instances stored in the `manager` column are nonreferenceable. You can only use the SQL interface to fetch `manager` column instances. But if `employee_t` has any `REF` attributes, OCI calls can then be used to navigate the `REF`.

**See Also:**

- OCIObjectPin()
- OCICacheFlush()

## 19.3.13 Pin Count and Unpinning

Each object in the object cache has a *pin count* associated with it.

The pin count 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 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:**

- About Freeing an Object Copy for more information about the conditions under which objects with zero pin count are removed from the cache and about objects being aged out of the cache
- About Marking Objects and Flushing Changes for information about explicitly flushing an object or the entire cache
- OCIObjectUnpin()
- OCIObjectPinCountReset()
- OCICacheUnpin()

## 19.3.14 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 `NULL`s 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*, meaning 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 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. Example 19-6 shows the C representations of types with their corresponding `NULL` indicator structures.

> **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()` 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;
```

### Example 19-6    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;
```

**See Also:**

- Using the Object Type Translator with OCI for more information about OTT-generated `NULL` indicator structures
- OCICollGetElem()
- OCIObjectGetInd()

## 19.3.15 About 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 resides. This value can be retrieved by calling `OCIObjectPinTable()`, and it is passed in the `table` parameter. To create a transient object, the application must 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.

This section includes the following topic: Attribute Values of New Objects.

**See Also:**

- OCIObjectNew()
- OCIObjectPinTable()
- OCIDescribeAny()

### 19.3.15.1 Attribute Values of New Objects

By default, all attributes of a newly created object 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. The attributes of the object have the default values described in Table 19-4, and the corresponding `NULL` indicators are set to `NOT NULL`.

### Table 19-4 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 that Oracle Database allows, which is midnight, 01-JAN-4712 BCE (equivalent to Julian day 1) |
| `ANSI DATE` | The earliest possible date that Oracle Database allows, 01-JAN-4712 BCE (equivalent to Julian day 1) |
| `TIMESTAMP` | The earliest possible date and time that Oracle Database allows, which is midnight, 01-JAN-4712 BCE (equivalent to Julian day 1) |
| `TIMESTAMP WITH TIME ZONE` | The earliest possible date and time that Oracle Database 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 that Oracle Database 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 file name. |

**See Also:**

- OCIAttrSet()
- OCIObjectNew()

## 19.3.16 About 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).

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 and 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:**

- Memory Layout of an Instance
- OCI Navigational and Type Functions
- OCIObjectNew()
- OCIObjectFree()
- OCIObjectMarkDelete()
- OCIObjectCopy()

## 19.3.17 Object Reference and Type Reference

The object extensions to OCI provide the application with the flexibility to access the contents of objects using their pointers or their references.

OCI provides the function `OCIObjectGetObjectRef()` to return a reference to an object when given the object's pointer.

For applications that also want to access the type information of objects, OCI provides the function `OCIObjectGetProperty()` to return a reference to an object's type descriptor object (TDO), when given a pointer to the object.

When a persistent object based on an object table with system-generated object identifiers (OIDs) is created, a reference to this object may be immediately obtained by using `OCIObjectGetObjectRef()`. But when a persistent object is based on an object view or on an object table with primary-key-based OIDs, all attributes belonging to the primary key must first be set before a reference can be obtained.

**See Also:**

- OCIObjectGetObjectRef()
- OCIObjectGetProperty()
- OCIObjectGetObjectRef()

## 19.3.18 Create Objects Based on Object Views and Object Tables with Primary-Key-Based OIDs

Applications can use the `OCIObjectNew()` call to create objects, which are based on object views, or on object tables with primary-key-based object identifiers (OIDs).

Because object identifiers of such views and tables are based on attribute values, applications must then use `OCIObjectSetAttr()` to set all attributes belonging to the primary key. Once the attribute values have been set, applications can obtain an object reference based on the attribute value by calling `OCIObjectGetObjectRef()`.

This process involves the following steps:

1. Pin the object view or object table on which the new object is to 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. Use `OCIObjectSetAttr()` to fill in the necessary values for the object attributes. These must include those attributes that make up the user-defined object identifier for the object table or object view.
4. Use `OCIObjectNew()` to allocate an object reference, passing in the handle to the table or view obtained by the pin operation in Step 1.
5. Use `OCIObjectGetObjectRef()` to obtain the primary-key-based reference to the object, if necessary. If desired, return to Step 2 to create more objects.
6. Flush the newly created objects to the server.

**Example 19-7    Creating a New Object for an Object View**

```
void object_view_new ()
{
void     *table;
OCIRef   *pkref;
void     *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"),(void  *) 0, OCI_DURATION_SESSION, (void  **) &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, (void  *)0,
    OCI_DURATION_SESSION,TRUE,&pkref);

/* Get the reference using OCIObjectGetObjectRef */
OCIObjectGetObjectRef(envp,errorp,object,pkref);
...
/* Flush new objects to server */
...
} /* end function */
```

Example 19-7 shows how this process might be implemented to create a new object for the `emp_view` object view in the `HR` schema.

**See Also:**

- OCIObjectNew()
- OCIObjectSetAttr()
- OCIObjectGetObjectRef()

### 19.3.19 Error Handling in Object Applications

Is like any other OCI application.

Error handling in OCI applications is the same whether or not the application uses objects.

**See Also:**

Error Handling in OCI for more information about function return codes and error messages

## 19.4 About 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.

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 Oracle Database 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));
```

See OTT Support for Type Inheritance for the resulting structs generated by OTT for this example.

This subtype `Student_t` can have its own subtype, such as `PartTimeStudent_t`:

```
CREATE TYPE PartTimeStudent_t UNDER Student_t
( numhours NUMBER) ;
```

This section includes the following topics:

- Substitutability
- NOT INSTANTIABLE Types and Methods
- OCI Support for Type Inheritance
- OTT Support for Type Inheritance

**See Also:**

*Oracle Database Object-Relational Developer's Guide* for a more complete discussion about type inheritance

## 19.4.1 Substitutability

Object type attributes and collection element types are substitutable.

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 you 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))
```

## 19.4.2 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 is not possible to construct instances of this type. The typical usage would be to 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.

## 19.4.3 OCI Support for Type Inheritance

Lists the calls that support type inheritance.

The following calls support type inheritance:

- OCIDescribeAny()
- Bind and Define Functions
- OCIObjectGetTypeRef()
- OCIObjectCopy()
- OCICollAssignElem()
- OCICollAppend()
- OCICollGetElem()

### 19.4.3.1 OCIDescribeAny()

The `OCIDescribeAny()` function provides 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 8-7 and Table 8-9 for attributes that `OCIDescribeAny()` can use to describe existing schema and subschema objects
> - OCIDescribeAny()

### 19.4.3.2 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, because all type checking and conversions are done on the server side.

OCI define functions also support substitutability (subtype instances can be fetched into define variables declared to hold the supertype). However, 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 resizable) 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:**
>
> Object-Relational Data Types in OCI for details of the bind and define processes

### 19.4.3.3 OCIObjectGetTypeRef()

The `OCIObjectGetTypeRef()` function returns the `REF` of the TDO of the most specific type of the input object.

This operation returns an error if the user does not have privileges on the most specific type.

> **See Also:**
>
> OCIObjectGetTypeRef()

### 19.4.3.4 OCIObjectCopy()

The `OCIObjectCopy()` 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()

### 19.4.3.5 OCICollAssignElem()

The input element can be an instance of the subtype of the declared type.

If the collection is of type `Person_t`, you can use the `OCICollAssignElem()` function to assign an `Employee_t` instance as an element of the collection.

> **See Also:**
>
> OCICollAssignElem()

### 19.4.3.6 OCICollAppend()

The input element can be an instance of the subtype of the declared type.

If the collection is of type `Person_t`, you can use the `OCICollAppend()` function to append an `Employee_t` instance to the collection.

> **See Also:**
>
> OCICollAppend()

### 19.4.3.7 OCICollGetElem()

The collection element returned could be an instance of the subtype of the declared type.

If the collection is of type `Person_t`, you can use the `OCICollGetElem()` function to get a pointer to an element, such as an `Employee_t` instance, in this collection.

> **See Also:**
>
> OCICollGetElem()

## 19.4.4 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 for an example and discussion

# 19.5 About Type Evolution

Adding, dropping, and modifying type attributes are supported. This concept is known as *type evolution*.

It is discussed in the *Oracle Database Object-Relational Developer's Guide*.

`OCIDescribeAny()` returns 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, if the name input to `OCIDescribeAny()` is a type name.

> **See Also:**
>
> - OCITypeArrayByName() and OCITypeByName(). To access type information, use these functions and `OCIDescribeAny()`
> - Type Evolution and the Object Cache for a discussion of the effect of type evolution on the object cache
> - OCIDescribeAny()

## 20

# Object-Relational Data Types in OCI

This chapter describes the purpose and structure of each of the data types that can be manipulated by the OCI data type mapping and manipulation functions.

This chapter also summarizes the different function groups giving lists of available functions and their purposes. In addition, provides information about how to use these data types in bind and define operations within an OCI application.

This chapter contains these topics:

- Overview of OCI Functions for Objects
- About Mapping Oracle Data Types to C
- About Manipulating C Data Types with OCI
- Date (OCIDate)
- Datetime and Interval (OCIDateTime, OCIInterval)
- Number (OCINumber)
- Fixed or Variable-Length String (OCIString)
- Raw (OCIRaw)
- Collections (OCITable, OCIArray, OCIColl, OCIIter)
- About Multilevel Collection Types
- REF (OCIRef)
- Object Type Information Storage and Access
- AnyType, AnyData, and AnyDataSet Interfaces
- About Binding Named Data Types
- About Defining Named Data Types
- About Binding and Defining Oracle C Data Types
- SQLT_NTY Bind and Define Examples

## 20.1 Overview of OCI Functions for Objects

The OCI data type mapping and manipulation functions provide the ability to manipulate instances of predefined Oracle C data types.

These data types are used to represent the attributes of user-defined data types, including object types in Oracle Database.

Each group of functions within OCI is distinguished by a particular naming convention. The data type 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 data type, as in `OCIDateFromText()` and `OCIRawSize()`. As will be explained later, the names can be further subdivided into function groups that operate on a particular type of data.

The predefined Oracle C types on which these functions operate are also distinguished by names that begin with the prefix *OCI*, as in `OCIDate` or `OCIString`.

The data type mapping and manipulation functions are used when an application must manipulate, bind, or define attributes of objects that are stored in an Oracle database, or that have been retrieved by a SQL query. Retrieved objects are stored in the client-side object cache.

The OCI client must allocate a descriptor before performing a bind or define operation. `OCIStmtExecute()` and `OCIStmtFetch2()` cannot 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 OCI in object mode and creating an OCI application that accesses and manipulates objects, see About Initializing the Environment and the Object Cache.

> **Note:**
>
> Operations on object types such as `OCIDate`, allow the address of the result to be the same as that of one of the operands.

> **See Also:**
>
> - OCIDateFromText()
> - OCIRawSize()
> - Object Advanced Topics in OCI for more information about retrieved objects being stored in the client-side object cache
> - OCIStmtExecute()
> - OCIStmtFetch2()
> - OCIDescriptorAlloc()
> - *Oracle Database Object-Relational Developer's Guide* for detailed information about object types, attributes, and collection data types

## 20.2 About Mapping Oracle Data Types to C

Oracle provides a rich set of predefined data types with which you can create tables and specify user-defined data types (including object types).

Object types extend the functionality of Oracle Database by allowing you to create data types 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.

You can use `NCHAR` and `NVARCHAR2` as attributes in objects and map to `OCIString *` in C.

Database tables and object types are based upon the data types supplied by Oracle. These tables and types are created with SQL statements and stored using a specific set of Oracle internal data types, like `VARCHAR2` or `NUMBER`. For example, the following SQL statements create a user-defined `address` data type 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 data types that represent C language mappings of Oracle data types.

An additional C type, `OCIInd`, is used to represent null indicator information corresponding to attributes of object types.

This section includes the following topic: OCI Type Mapping Methodology.

**See Also:**

- Table 24-1 for the available Oracle types and their C mappings you can use as object attribute types
- Using the Object Type Translator with OCI for more information and examples about using OTT

## 20.2.1 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 data types like `OCINumber` is opaque to client applications, and the data types are manipulated with a set of predefined functions. This allows 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 writing a C program to manipulate Oracle number variables without losing the accuracy provided by Oracle numbers. To do this operation in Oracle Database Release 7, you would have had to issue a "`SELECT ... FROM DUAL`" statement. In later releases, this is accomplished by invoking the `OCINumber*()` functions.

## 20.3 About Manipulating C Data Types with OCI

Shows how you can manipulate C data types with OCI and shows the various function prefixes, along with example function names and the data types on which the functions operate.

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 such as `integer`. However, the C data types listed in Table 24-1 are not simple C primitives. Types such as `OCIString` and `OCINumber` are actually structs with a specific Oracle-defined internal structure. It is not possible to simply add together two `OCINumbers` and store the value in the third.

The following is not valid:

```
OCINumber    num_1, num_2, sum;
...
/* some initialization occurs */
...
sum = num_1 + num_2;           /* NOT A VALID OPERATION */
```

The OCI data type mapping and manipulation functions are provided to enable you to perform operations on these new data types. For example, the preceding addition of `OCINumbers` could be accomplished as follows, using the `OCINumberAdd()` function:

```
OCINumber    num_1, num_2, sum;
...
/* some initialization occurs */
...
OCINumberAdd(errhp, &num_1, &num_2, &sum): /* errhp is error handle */
```

OCI provides functions to operate on each of the new data types. The names of the functions provide information about the data types on which they operate. The first three letters, *OCI*, indicate that the function is part of OCI. The next part of the name indicates the data type on which the function operates. Table 20-1 shows the various function prefixes, along with example function names and the data types on which the functions operate.

**Table 20-1 Function Prefix Examples**

| Function Prefix | Example | Operates on |
|---|---|---|
| `OCIColl` | OCICollGetElem() | `OCIColl, OCIIter, OCITable, OCIArray` |
| `OCIDate` | OCIDateDaysBetween() | `OCIDate` |
| `OCIDateTime` | OCIDateTimeSubtract() | `OCIDate, OCIDateTime` |
| `OCIInterval` | OCIIntervalToText() | `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 data types is described later in this chapter, along with a list of the functions that manipulate that type.

This section includes the following topic: Precision of Oracle Number Operations.

## 20.3.1 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.

# 20.4 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` data type 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:**
>
> OCI Data Type Mapping and Manipulation Functions for the prototypes and descriptions of all the functions

This section includes the following topic: Date Example.

## 20.4.1 Date Example

Shows how to manipulate an attribute of type `OCIDate` using OCI calls.

Example 20-1 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.

The output is:

```
For: FRIDAY    , OCTOBER   05, 1990
The last day of the month is: WEDNESDAY, OCTOBER   31, 1990
The next Wednesday is: WEDNESDAY, OCTOBER   10, 1990
```

**Example 20-1 Manipulating an Attribute of Type OCIDate**

```
#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 */
/* 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 the information */
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);
```

**See Also:**

- OCI Environment Initialization for information about initializing `OCIEnv` and `OCIError`
- Object Cache Operations for information about pinning.

## 20.5 Datetime and Interval (OCIDateTime, OCIInterval)

The `OCIDateTime` data type is an opaque structure used to represent Oracle time-stamp data types (`TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`) and the `ANSI DATE` data type.

You can set or retrieve the data in these types (that is, year, day, fractional second) using the appropriate OCI functions.

The `OCIInterval` data type is also an opaque structure and is used to represent Oracle interval data types (`INTERVAL YEAR TO MONTH`, `INTERVAL DAY TO SECOND`).

You can bind and define `OCIDateTime` and `OCIInterval` data using the following external typecodes shown in Table 20-2 in the bind or define call.

**Table 20-2 Binding and Defining Datetime and Interval Data Types**

| OCI Data Type | Type of Data | External Typecode for Binding/ Defining |
|---|---|---|
| `OCIDateTime` | ANSI DATE | SQLT_DATE |
| `OCIDateTime` | TIMESTAMP | SQLT_TIMESTAMP |
| `OCIDateTime` | TIMESTAMP WITH TIME ZONE | SQLT_TIMESTAMP_TZ |
| `OCIDateTime` | TIMESTAMP WITH LOCAL TIME ZONE | SQLT_TIMESTAMP_LTZ |
| `OCIInterval` | INTERVAL YEAR TO MONTH | SQLT_INTERVAL_YM |
| `OCIInterval` | INTERVAL DAY TO SECOND | SQLT_INTERVAL_DS |

In general, functions that operate on `OCIDateTime` data are also valid for `OCIDate` data.

This section includes the following topics:

- About Datetime Functions
- Datetime Example
- About Interval Functions

**See Also:**

- Table 20-3 and Table 20-4 for information about OCI functions that operate on datetime and interval data
- OCI Date, Datetime, and Interval Functions for more detailed information about datetime and interval functions

## 20.5.1 About Datetime Functions

Lists and describes functions that operate on `OCIDateTime` values.

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 listed in Table 20-3 for more information about input types that are valid for a particular function.

**Table 20-3 Datetime Functions**

| Function | Purpose |
|---|---|
| OCIDateTimeAssign() | Performs datetime assignment |
| OCIDateTimeCheck() | Checks if the given date is valid |
| OCIDateTimeCompare() | Compares two datetime values |
| OCIDateTimeConstruct() | Constructs a datetime descriptor |
| OCIDateTimeConvert() | Converts one datetime type to another |
| OCIDateTimeFromArray() | Converts an array containing a date to an `OCIDateTime` descriptor |
| OCIDateTimeFromText() | Converts the given string to Oracle datetime type in the `OCIDateTime` descriptor, according to the specified format |
| OCIDateTimeGetDate() | Gets the date (year, month, day) portion of a datetime value |
| OCIDateTimeGetTime() | Gets the time (hour, minute, second, fractional second) from datetime value |
| OCIDateTimeGetTimeZoneName() | Gets the time zone name portion of a datetime value |
| OCIDateTimeGetTimeZoneOffset() | Gets the time zone (hour, minute) portion of a datetime value |
| OCIDateTimeIntervalAdd() | Adds an interval to a datetime to produce a resulting datetime |
| OCIDateTimeIntervalSub() | Subtracts an interval from a datetime and stores the result in a datetime |
| OCIDateTimeSubtract() | Takes two datetimes as input and stores their difference in an interval |
| OCIDateTimeSysTimeStamp() | Gets the system current date and time as a time stamp with time zone |
| OCIDateTimeToArray() | Converts an `OCIDateTime` descriptor to an array |

**Table 20-3 (Cont.) Datetime Functions**

| Function | Purpose |
|---|---|
| OCIDateTimeToText() | Converts the given date to a string according to the specified format |
| OCIDateZoneToZone() | Converts the date from one time zone to another time zone |

## 20.5.2 Datetime Example

Shows how to use an `OCIDateTime` data type to select data from a `TIMESTAMP WITH LOCAL TIME ZONE` column.

The code fragment in Example 20-2 shows how to use an `OCIDateTime` data type to select data from a `TIMESTAMP WITH LOCAL TIME ZONE` column.

**Example 20-2 Manipulating an Attribute of Type OCIDateTime**

```
...

/* allocate the program variable for storing the data */
OCIDateTime *tstmpltz = (OCIDateTime *)NULL;

/* Col1 is a time stamp with local time zone column */
OraText *sqlstmt = (OraText *)"SELECT col1 FROM foo";

/* Allocate the descriptor (storage) for the data type */
status = OCIDescriptorAlloc(envhp,(void  **)&tstmpltz, OCI_DTYPE_TIMESTAMP_LTZ,
         0, (void  **)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 time stamp 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);
...
```

## 20.5.3 About Interval Functions

Lists and describes the interval functions.

The functions listed in Table 20-4 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.

**Table 20-4 Interval Functions**

| Function | Purpose |
|---|---|
| OCIIntervalAdd() | Adds two intervals to produce a resulting interval |
| OCIIntervalAssign() | Copies one interval to another |
| OCIIntervalCheck() | Checks the validity of an interval |
| OCIIntervalCompare() | Compares two intervals |
| OCIIntervalDivide() | Divides an interval by an Oracle `NUMBER` to produce an interval |
| OCIIntervalFromNumber() | Converts an Oracle `NUMBER` to an interval |
| OCIIntervalFromText() | When given an interval string, converts the interval represented by the string |
| OCIIntervalFromTZ() | Returns an interval when given an input string of time zone form |
| OCIIntervalGetDaySecond() | Gets values of day, hour, minute, and second from an interval |
| OCIIntervalGetYearMonth() | Gets year and month from an interval |
| OCIIntervalMultiply() | Multiplies an interval by an Oracle `NUMBER` to produce an interval |
| OCIIntervalSetDaySecond() | Sets day, hour, minute, and second in an interval |
| OCIIntervalSetYearMonth() | Sets year and month in an interval |
| OCIIntervalSubtract() | Subtracts two intervals and stores the result in an interval |
| OCIIntervalToNumber() | Converts an interval to an Oracle `NUMBER` |
| OCIIntervalToText() | When given an interval, produces a string representing the interval |

**See Also:**

OCI Date, Datetime, and Interval Functions for complete lists of the names and purposes and more detailed information about these functions

# 20.6 Number (OCINumber)

The `OCINumber` data type is an opaque structure used to represent Oracle numeric data types (`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`.

This section includes the following topic: OCINumber Examples.

**See Also:**

Table 28-11for the prototypes and descriptions for all the OCI `NUMBER` functions

## 20.6.1 OCINumber Examples

Shows how to manipulate an attribute of type `OCINumber`, how to convert values in `OCINumber` format returned from `OCIDescribeAny()` calls to unsigned integers, and 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.

The code fragment in Example 20-3 shows how to manipulate an attribute of type `OCINumber`. The code fragment in Example 20-4 shows how to convert values in `OCINumber` format returned from `OCIDescribeAny()` calls to unsigned integers.

Example 20-4 shows 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 20-3    Manipulating an Attribute of Type OCINumber**

```
/* 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 that 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 the 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 */
```

**Example 20-4    Converting Values in OCINumber Format Returned from OCIDescribeAny() Calls to Unsigned Integers**

```
/* 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 ((void *)_dschp, (ub4)OCI_HTYPE_DESCRIBE, &parmdp, 0,
                         (ub4)OCI_ATTR_PARAM, _errhp);
if (isError (status, _errhp))
{
return 0;
}
status = OCIAttrGet ((void *)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);
```

## 20.7 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`.

This section includes the following topics:

- About String Functions
- String Example

**See Also:**

Table 28-16 for the prototypes and descriptions for all the string functions

## 20.7.1 About String Functions

Lists and describes functions that allow the C programmer to manipulate an instance of a string.

Table 20-5 shows the functions that allow the C programmer to manipulate an instance of a string.

**Table 20-5    String Functions**

| Function | Purpose |
|---|---|
| OCIStringAllocSize() | Get allocated size of string memory in code points (Unicode) or bytes |
| OCIStringAssign() | Assign one string to another string |
| OCIStringAssignText() | Assign the source text string to the target string |
| OCIStringPtr() | Get a pointer to the text of a given string |
| OCIStringResize() | Resize the memory of a given string |
| OCIStringSize() | Get the size of a given string |

## 20.7.2 String Example

Assigns a text string to a string, then gets a pointer to the string part of the string, and the string size, and prints it out.

Example 20-5 assigns a text string to a string, then gets a pointer to the string part of the string, and the string size, and prints it out.

Note the double indirection used in passing the *`vstring1`* parameter in `OCIStringAssignText()`.

**Example 20-5    Manipulating an Attribute of Type OCIString**

```
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);
```

> **See Also:**
>
> OCIStringAssignText()

## 20.8 Raw (OCIRaw)

Variable-length raw data is represented in C using the `OCIRaw *` data type.

For binding and defining variables of type `OCIRaw *`, use the external typecode `SQLT_LVB`.

This section includes the following topics:

- About Raw Functions
- Raw Example

> **See Also:**
>
> Table 28-14 for the prototypes and descriptions for all the raw functions

### 20.8.1 About Raw Functions

Lists and describes the Raw functions.

Table 20-6 shows the functions that perform OCIRaw operations.

**Table 20-6 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 |

### 20.8.2 Raw Example

Shows how to set up a raw data block and obtain a pointer to its data.

Example 20-6 shows how to set up a raw data block and obtain a pointer to its data.

Note the double indirection in the call to `OCIRawAssignBytes()`.

**Example 20-6    Manipulating an Attribute of Type OCIRaw**

```
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);
```

**See Also:**

OCIRawAssignBytes()

## 20.9 Collections (OCITable, OCIArray, OCIColl, OCIIter)

Oracle Database 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 data types (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.

This section includes the following topics:

- Generic Collection Functions
- About Collection Data Manipulation Functions
- About Collection Scanning Functions
- Varray/Collection Iterator Example
- About Nested Table Manipulation Functions
- About Nested Table Manipulation Functions

You can allocate an instance of a varray or nested table using `OCIObjectNew()` and free it using `OCIObjectFree()`.

**See Also:**

- About Nested Table Manipulation Functions
- OCIObjectNew()
- OCIObjectFree()
- OCI Collection and Iterator Functions for the prototypes and descriptions for these functions

## 20.9.1 Generic Collection Functions

Oracle Database provides two types of collections: variable-length arrays (varrays) and nested tables.

Both varrays and nested tables can be viewed as subtypes of a generic collection type.

In C, a generic collection is represented as `OCIColl *`, a varray is represented as `OCIArray *`, and a nested table is represented 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()`.

**Note:**

Indexes passed to collection functions are zero-based.

**See Also:**

- OCITableExists()
- About Nested Table Manipulation Functions

## 20.9.2 About Collection Data Manipulation Functions

Lists and describes the collection data manipulation functions.

Table 20-7 shows the generic functions that manipulate collection data.

**Table 20-7    Collection Functions**

| Function | Purpose |
|---|---|
| OCICollAppend() | Append an element to the end of a collection |
| OCICollAssign() | Assign one collection to another |
| OCICollAssignElem() | Assign element at given index |
| OCICollGetElem() | Get pointer to an element when given its index |
| OCICollGetElemArray() | Get array of elements from a collection |
| OCICollIsLocator() | Indicate 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 |

## 20.9.3 About Collection Scanning Functions

Lists and describes the collection scanning functions.

Table 20-8 shows the generic functions that enable you to scan collections with a collection iterator. The iterator is of type `OCIIter`, and is created by first calling `OCIIterCreate()`.

**Table 20-8    Collection Scanning Functions**

| Function | Purpose |
|---|---|
| OCIIterCreate() | Create an iterator to scan the elements of a collection |
| OCIIterDelete() | Delete a collection iterator |
| OCIIterGetCurrent() | Get a pointer to the current element pointed to by the iterator |
| OCIIterInit() | Initialize an iterator to scan the given collection |
| OCIIterNext() | Get a pointer to the next iterator collection element |
| OCIIterPrev() | Get pointer to the previous iterator collection element |

## 20.9.4 Varray/Collection Iterator Example

Shows how to create and use a collection iterator to scan through a varray.

Example 20-7 creates and uses a collection iterator to scan through a varray.

**Example 20-7    Using Collection Data Manipulation Functions**

```
OCIEnv       *envhp;
OCIError     *errhp;
text         *text_ptr;
sword        status;
OCIArray     *clients;
OCIString    *client_elem;
OCIIter      *iterator;
boolean      eoc;
void         *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,
                    (void  **) &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(). See 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,
                      (void  **)&elemind, &eoc);
}

if (status != OCI_SUCCESS)
{
  /* handle error */
}

/* destroy the iterator */
status = OCIIterDelete(envhp, errhp, &iterator);
```

## 20.9.5 About 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 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, Table 20-9 shows the functions that are specific to nested tables. They should not be used on varrays.

**Table 20-9 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 the index for the first existing element of a table |
| OCITableLast() | Return the index for the last existing element of a table |
| OCITableNext() | Return the index for the next existing element of a table |
| OCITablePrev() | Return the index for the previous existing element of a table |
| OCITableSize() | Return the table size, not including any deleted elements |

This section includes the following topic: Nested Table Element Ordering.

### 20.9.5.1 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.

### 20.9.6 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 that 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; it only locates 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.

You can use the `OCICollIsLocator()` function to determine whether a collection is locator-based or not.

**See Also:**

- OCICollIsLocator()
- *Oracle Database SQL Language Reference*

## 20.10 About 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` an `OCIColl*`, which you can use 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()`

This section includes the following topic: Multilevel Collection Type Example.

**See Also:**

- OCICollGetElem()
- OCIIterGetCurrent()
- OCIIterNext()
- OCIIterPrev()
- OCICollAssignElem()
- OCICollAppend()

## 20.10.1 Multilevel Collection Type Example

Shows how to iterate over the multilevel collection.

The following types and tables are used for Example 20-8.

```
type_1 (a NUMBER, b NUMBER)
NT1 TABLE OF type_1
NT2 TABLE OF NT1
```

The code fragment in Example 20-8 iterates over the multilevel collection.

**Example 20-8 Using Multilevel Collection Data Manipulation Functions**

```
...
OCIColl *outer_coll;
OCIColl *inner_coll;
OCIIter *itr1, *itr2;
Type_1 *type_1_instance;
..
/* assume that 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, (void  **) &elem,
                               (void  **) &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, (void  **)&elem2,
              (void  **) &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));
...
```

## 20.11 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*`.

This section includes the following topic:

- About REF Manipulation Functions
- REF Example

> **See Also:**
>
> Table 28-15 for the prototypes and descriptions for all the `REF` manipulation functions

### 20.11.1 About REF Manipulation Functions

Lists and describes `REF` manipulation functions.

Table 20-10 shows the functions that perform `REF` operations.

**Table 20-10 REF Manipulation Functions**

| Function | Purpose |
|---|---|
| OCIRefAssign() | Assign one REF to another |
| OCIRefClear() | Clear or nullify a REF |
| OCIRefFromHex() | Convert a hexadecimal string to a REF |
| OCIRefHexSize() | Return the size of a hexadecimal string representation of REF |
| OCIRefIsEqual() | Compare two REFs for equality |
| OCIRefIsNull() | Test whether a REF is `NULL` |
| OCIRefToHex() | Convert a REF to a hexadecimal string |

## 20.11.2 REF Example

Shows how to test two `REF`s for `NULL`, compares them for equality, and assigns one `REF` to another.

Example 20-9 tests two `REF`s for `NULL`, compares them for equality, and assigns one `REF` to another. Note the double indirection in the call to `OCIRefAssign()`.

**Example 20-9 Using REF Manipulation Functions**

```
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");
```

**See Also:**

OCIRefAssign()

# 20.12 Object Type Information Storage and Access

The OCI data types and type descriptors are discussed in this section.

This section includes the following topic: Descriptor Objects.

## 20.12.1 Descriptor Objects

Lists and describes the 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. Table 20-11 lists the OCI data types associated with each type of descriptor object.

**Table 20-11 Descriptor Objects**

| Information Type | OCI Data Type |
|---|---|
| Type | OCIType |
| Type Attributes Collection Elements Method Parameters 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.

**See Also:**

- OCIBindObject()
- OCIObjectNew()
- OCITypeByName()

## 20.13 AnyType, AnyData, and AnyDataSet Interfaces

The AnyType, AnyData, and AnyDataSet interfaces allow you to model self-descriptive data.

You can store heterogeneous data types in the same column and query the type of data in an application.

These definitions are used in the discussion in the following sections:

- *Persistent types.* Types that 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 as needed. 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` data type models such data in OCI. A data value of most SQL types can be converted to an OCIAnyData that 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` data type 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) and in SQL and PL/SQL for constructing and manipulating these type descriptions and self-descriptive data. The following sections describe the relevant OCI interfaces.

This section includes the following topics:

- About Type Interfaces
- About OCIAnyData Interfaces
- NCHAR Typecodes for OCIAnyData Functions
- About OCIAnyDataSet Interfaces

**See Also:**

- Persistent Objects, Transient Objects, and Values
- *Oracle Database SQL Language Reference* for an overview in the section about Oracle-supplied types

## 20.13.1 About Type Interfaces

You can use the type interfaces to construct named and anonymous transient object types (structured with attributes) and collection types.

Use the `OCITypeBeginCreate()` call to begin type construction of transient object types and collection types (the typecode parameter determines which one is being constructed).

You must allocate a parameter handle using `OCIDescriptorAlloc()`. Subsequently, you set type information (for attributes of an object type and for the collection element's type) by using `OCIAttrSet()`. For object types, as shown in Example 20-10, use `OCITypeAddAttr()` to add the attribute information to the type. After adding information for the last attribute, you must call `OCITypeEndCreate()`.

For collection types, as shown in Example 20-11, use `OCITypeSetCollection()` to set the information on the collection element type. Subsequently, call `OCITypeEndCreate()` to finish construction.

You can use the `OCIDescribeAny()` call to obtain the `OCIType` corresponding to a persistent type.

**Example 20-10    Using Type Interfaces to Construct Object Types**

```
OCITypeBeginCreate( ...)        /* Begin Type Creation */
OCIDescriptorAlloc(...)
OCIAttrSet(...)
OCITypeAddAttr(...)             /* Add attribute 1 */
OCIAttrSet(...)
OCITypeAddAttr(...)             /* Add attribute 2 */
...
OCITypeEndCreate(...)           /* End Type Creation */
```

**Example 20-11 Using Type Interfaces to Construct Collection Types**

```
OCITypeBeginCreate( ...)        /* Begin Type Creation */
OCIDescriptorAlloc(...)
OCIAttrSet(...)
OCITypeSetCollection(...)       /* Set information on collection element */
OCITypeEndCreate(...)           /* End Type Creation */
```

This section includes the following topics:

- About Creating a Parameter Descriptor for OCIType Calls
- About Obtaining the OCIType for Persistent Types
- Type Access Calls
- Extensions to OCIDescribeAny()

**See Also:**

- OCITypeBeginCreate()
- OCIDescriptorAlloc()
- OCIAttrSet()
- OCITypeAddAttr()
- OCITypeEndCreate()
- OCITypeSetCollection()
- OCITypeEndCreate()

## 20.13.1.1 About Creating a Parameter Descriptor for OCIType Calls

You can use the `OCIDescriptorAlloc()` call to allocate an `OCIParam` (with the parent handle being the environment handle).

Subsequently, you can call `OCIAttrSet()` 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 the precision value.

- `OCI_ATTR_SCALE`

To set numeric scale. Pass a `(sb1 *)` attribute value to the buffer that is holding the scale value.

- `OCI_ATTR_CHARSET_ID`

To set the character set ID for character types. Pass a `(ub2 *)` attribute value to the buffer holding the 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 the character set form value.

- `OCI_ATTR_DATA_SIZE`

Length of `VARCHAR2`, `RAW`, and so on. Pass a `(ub2 *)` attribute value to the buffer holding the length.

- `OCI_ATTR_TYPECODE`

To set typecode. Pass a `(ub2 *)` attribute value to the buffer holding the typecode. This attribute must be set first.

- `OCI_ATTR_TDO`

To set `OCIType` of an object or collection attribute. Pass an `(OCIType *)` attribute value to the `OCIType` corresponding to the attribute. Ensure 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. Otherwise, an exception is returned.

- For built-in types, the following typecodes are acceptable (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`, and `OCI_TYPECODE_INTERVAL_DS`.

- If the attribute or 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`.

**See Also:**

- OCIDescriptorAlloc()
- OCIAttrSet()

### 20.13.1.2 About Obtaining the OCIType for Persistent Types

You can use the `OCIDescribeAny()` call to obtain the `OCIType` corresponding to a persistent type.

This is shown in the following example:

```
OCIDescribeAny(svchp, errhp. (void  *)"HR.EMPLOYEES",
               (ub4)strlen("HR.EMPLOYEES"),
               (ub1)OCI_OTYPE_NAME, (ub1)OCI_DEFAULT, OCI_PTYPE_TYPE, dschp);
```

From the describe handle (*dschp*), you can use `OCIAttrGet()` calls to obtain the `OCIType`.

> **See Also:**
>
> - OCIDescribeAny()
> - OCIAttrGet()

### 20.13.1.3 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 and position.

> **See Also:**
>
> OCIDescribeAny()

### 20.13.1.4 Extensions to OCIDescribeAny()

For transient types that represent built-in types (created with a built-in typecode), the parameter handle that describes these types (which are of type `OCI_PTYPE_TYPE`) .

The parameter handle that describes these transient types supports 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 have the usual meanings they have while describing a type attribute.

> **Note:**
>
> These attributes are supported only for transient built-in 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 are of type `OCI_PTYPE_TYPE_ATTR`).

## 20.13.2 About OCIAnyData Interfaces

An `OCIAnyData` encapsulates type information and a data instance of that type (that is, self-descriptive data).

An `OCIAnyData` can be created from any built-in or user-defined type instance by 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) by calling `OCIAnyDataBeginCreate()` with the type information (`OCIType`). Subsequently, you can use `OCIAnyDataAttrSet()` for object types and use `OCIAnyDataCollAddElem()` for collection types. Finally, use the `OCIAnyDataEndCreate()` call to finish the construction process.

Subsequently, you can invoke the access routines. To convert (cast) an `OCIAnyData` to the corresponding type instance, you can use `OCIAnyDataAccess()`.

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. You can use these calls to avoid unnecessary creation and copying of the entire collection in memory, as shown in Example 20-12.

or

```
OCIAnyDataCollAddElem(...)    /* Element-wise construction for collections */

OCIAnyDataEndCreate(...)      /* End OCIAnyData Creation */
```

**Example 20-12 Using Special Construction and Access Calls for Improved Performance**

```
OCIAnyDataConvert(...)        /* Cast a built-in or user-defined type instance
                                 to an OCIAnyData in 1 call. */

OCIAnyDataBeginCreate(...)    /* Begin AnyData Creation */

OCIAnyDataAttrSet(...)        /* Attribute-wise construction for object types */
```

**See Also:**

- OCIAnyDataConvert()
- OCIAnyDataBeginCreate()
- OCIAnyDataAttrSet()
- OCIAnyDataCollAddElem()
- OCIAnyDataEndCreate()
- OCIAnyDataAccess()

### 20.13.3 NCHAR Typecodes for OCIAnyData Functions

The function `OCIAnyDataTypeCodeToSqlt()` converts the `OCITypeCode` for an OCIAnyData 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 character set form to determine if the data is `NCHAR` (if character set form is SQLCS_NCHAR).

`OCIAnyDataTypeCodeToSqlt()` converts `OCI_TYPECODE_CHAR` and `OCI_TYPECODE_VARCHAR2` to the output values `SQLT_VST` (which corresponds to the `OCIString` mapping) with a character set form of `SQLCS_IMPLICIT`. `OCI_TYPECODE_NVARCHAR2` also returns `SQLT_VST` (`OCIString` mapping is used by OCIAnyData API) with a character set form of `SQLCS_NCHAR`.

**See Also:**

- OCIDescribeAny()
- OCIAnyDataTypeCodeToSqlt()

### 20.13.4 About OCIAnyDataSet Interfaces

An `OCIAnyDataSet` encapsulates type information and *a set of instances* of that type. To begin the construction process, call `OCIAnyDataSetBeginCreate()`.

Call `OCIAnyDataSetAddInstance()` to add a new instance; this call returns the `OCIAnyData` corresponding to that instance.

Then, you can invoke the OCIAnyData functions to construct this instance. Call `OCIAnyDataSetEndCreate()` when all instances have been added.

For access, call `OCIAnyDataSetGetInstance()` to get the `OCIAnyData` corresponding to the instance. Only sequential access is supported. Subsequently, you can invoke the OCIAnyData access functions, as in the following 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 */
```

**See Also:**

- OCIAnyDataSetBeginCreate()
- OCIAnyDataSetAddInstance()
- OCIAnyDataSetEndCreate()
- OCIAnyDataSetGetInstance()
- OCI Any Type and Data Functions for complete descriptions of all the calls in these interfaces

# 20.14 About Binding Named Data Types

This section provides information about binding named data types (such as objects and collections) and `REF`s.

This section includes the following topics:

- Named Data Type Binds
- About Binding REFs
- Information for Named Data Type and REF Binds
- Information Regarding Array Binds

## 20.14.1 Named Data Type Binds

For a named data type (object type or collection) bind, a second bind call is necessary following `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`.

The `OCIBindObject()` call sets up additional attributes specific to the object type bind. An OCI application uses this call when fetching data from a table that has a column with an object data type.

The `OCIBindObject()` call takes, among other parameters, a type descriptor object (TDO) for the named data type. The TDO of data type `OCIType` is created and stored in the database when a named data type is created. It contains information about the type and its attributes. An application can obtain a TDO by calling `OCITypeByName()`.

The `OCIBindObject()` call also sets up the indicator variable or structure for the named data type bind.

When binding a named data type, use the `SQLT_NTY` data type constant to indicate the data type of the program variable being bound. `SQLT_NTY` indicates that a C struct representing the named data type is being bound. A pointer to this structure is passed to the bind call.

With inheritance and instance substitutability, you can bind a subtype instance where the supertype is expected.

Working with named data types may require the use of three bind calls in some circumstances. For example, to bind a static array of named data types to a PL/SQL table, three calls must be invoked: `OCIBindByName()` or `OCIBindByName2()`, `OCIBindArrayOfStruct()`, and `OCIBindObject()`.

**See Also:**

- Fetching Embedded Objects for information about using these data types to fetch an embedded object from the database
- Information for Named Data Type and REF Binds
- Descriptor Objects
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIBindObject()
- OCITypeByName()
- OCIBindArrayOfStruct()

## 20.14.2 About Binding REFs

As with named data types, binding `REF`s is a two-step process.

First, call `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`, and then call `OCIBindObject()`.

`REF`s are bound using the `SQLT_REF` data type. 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:**

- Retrieving an Object Reference from the Server for information about binding and pinning `REF`s to objects
- Information for Named Data Type and REF Binds for additional important information
- OCIBindByName() or OCIBindByName2()
- OCIBindByPos() or OCIBindByPos2()
- OCIBindObject()

## 20.14.3 Information for Named Data Type and REF Binds

Remember the following important information when you work with named data type and `REF` binds.

It includes pointers about memory allocation and indicator variable usage.

- If the data type being bound is `SQLT_NTY`, the indicator struct parameter of the `OCIBindObject()` call (`void ** indpp`) is used, and the scalar indicator is completely ignored.
- If the data type 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, therefore, 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 are optional. Users can pass `NULL` if these parameters are not needed.

**See Also:**

OCIBindObject()

## 20.14.4 Information Regarding Array Binds

For doing array binds of named data types or `REF`s, for array inserts or fetches, the user must 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 must be passed.

> **See Also:**
>
> Named Data Types: Object, VARRAY, Nested Table for more information about `SQLT_NTY`

## 20.15 About Defining Named Data Types

This section provides information about defining named data types (for example, objects, collections) and `REF`s.

This section includes the following topics:

- About Defining Named Data Type Output Variables
- About Defining REF Output Variables
- Information for Named Data Type and REF Defines, and PL/SQL OUT Binds

### 20.15.1 About Defining Named Data Type Output Variables

For a named data type (object type, nested table, varray) define, two define calls are necessary.

The application should first call `OCIDefineByPos()` or `OCIDefineByPos2()`, specifying `SQLT_NTY` in the *`dty`* parameter. Following `OCIDefineByPos()` or `OCIDefineByPos2()`, the application must call `OCIDefineObject()` to set up additional attributes pertaining to a named data type define. In this case, the data buffer pointer in `OCIDefineByPos()` or `OCIDefineByPos2()` is ignored.

Specify the `SQLT_NTY` data type constant for a named data type define. In this case, the application fetches the result data into a host-language representation of the named data type. In most cases, this is a C struct generated by the Object Type Translator.

To make 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, Oracle strongly recommends using objects in the object cache and *not* passing objects allocated out of user memory from the stack. Otherwise, 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 resize the object, which is possible only for objects in the cache.

> **See Also:**
>
> - Information for Named Data Type and REF Defines, and PL/SQL OUT Binds for more important information about defining named data types
> - OCIDefineByPos() or OCIDefineByPos2()
> - OCIDefineObject()
> - OCIObjectNew()

## 20.15.2 About Defining REF Output Variables

As with named data types, defining for a `REF` output variable is a two-step process.

The first step is a call to `OCIDefineByPos()` or `OCIDefineByPos2()`, and the second is a call to `OCIDefineObject()`. Also as with named data types, the `SQLT_REF` data type constant is passed to the *`dty`* parameter of `OCIDefineByPos()`.

`SQLT_REF` indicates that the application is fetching the result data into a variable of type `OCIRef *`. This `REF` can then be used as part of object pinning and navigation.

**See Also:**

- OCIDefineByPos() or OCIDefineByPos2()
- OCIDefineObject()
- Information for Named Data Type and REF Defines, and PL/SQL OUT Binds for more important information about defining `REF`s
- About Working with Objects in OCI for more information about object pinning and navigation

## 20.15.3 Information for Named Data Type and REF Defines, and PL/SQL OUT Binds

Consider the following important information as you work with named data type and `REF` defines. It includes pointers about memory allocation and indicator variable usage.

A PL/SQL OUT bind refers to binding a placeholder to an output variable in a PL/SQL block. Unlike a SQL statement, where output buffers are set up with define calls, in a PL/SQL block, output buffers are set up with bind calls.

- If the data type being defined is `SQLT_NTY`, then the indicator struct parameter of the `OCIDefineObject()` call (*`void ** indpp`*) is used, and the scalar indicator is completely ignored.
- If the data type is `SQLT_REF`, then 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, therefore, the user is not interested in any information about being null.
- 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 any secondary memory (out-of-line memory), is deallocated. The preallocated memory must come from the cache (the result of an `OCIObjectNew()` call).

> **Note:**
>
> If you want your client application to allocate memory from its own private memory space, instead of the cache, your application must ensure that there is no secondary out-of-line memory in the object.

To preallocate object memory for an object define with type `SQLT_NTY`, client applications must use the `OCIObjectNew()` function. A client application 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()`.

> **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 is 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 is 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 gets allocated (implicitly if necessary) and populated accordingly to indicate the atomic nullity of the object. The top-level object does not get implicitly allocated.
- An application can free indicators by calling `OCIObjectFree()`. If there is a top-level object (as with 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, *`indszp`*, and program variable size pointer, *`pvszsp`*, in the `OCIDefineObject()` call are optional. Users can pass `NULL` if these parameters are not needed.

This section includes the following topic: Information About Array Defines.

**See Also:**

- About Binding Placeholders in PL/SQL for more information
- OCIDefineObject()
- OCIObjectNew()
- OCIObjectFree()
- OCI Navigational and Type Functions for details about `OCIObjectNew()` and `OCIObjectFree()`
- OCIDefineObject()

### 20.15.3.1 Information About Array Defines

To perform array defines of named data types or `REF`s, the user must 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) must be passed.

## 20.16 About Binding and Defining Oracle C Data Types

This section summarizes information about binding and defining Oracle C named data types

Previous chapters of this book have discussed OCI bind and define operations. About Binding Placeholders in OCI discussed the basics of OCI bind operations, whereas About Defining Output Variables in OCI discussed the basics of OCI define operations. Information specific to binding and defining named data types and `REF`s was described in 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 data types and `REF`s showed how to bind or define an object or reference. Pinning an Object expanded on this to talk about pinning object references, Fetching Embedded Objects discussed fetching embedded instances, and Object Navigation discussed object navigation.

The purpose of this section is to cover binding and defining of individual attribute values, using the data type 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 because the appropriate data type code is specified. Table 20-12 lists the data types that you can use for binds and defines, along with their C mapping, and the OCI external data type that must be specified in the *dty* (data type code) parameter of the bind or define call.

**Table 20-12 Data Type Mappings for Binds and Defines**

| Data Type | C Mapping | OCI External Data Type 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 [1] |
| `RAW` | `OCIRaw *` | SQLT_LVB [1] |
| `CHAR`, `NCHAR` | `OCIString *` | SQLT_VST |
| Object | `struct *` | Named Data Type (SQLT_NTY) |
| `REF` | `OCIRef *` | REF (SQLT_REF) |
| `VARRAY` | `OCIArray *` | Named Data Type (SQLT_NTY) |
| Nested Table | `OCITable *` | Named Data Type (SQLT_NTY) |
| `DATETIME` | `OCIDateTime *` | See Datetime and Interval (OCIDateTime, OCIInterval). |
| `INTERVAL` | `OCIInterval *` | See Datetime and Interval (OCIDateTime, OCIInterval). |

[1] 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 `OCIStringResize()`.

The following sections presents examples of how to use C-mapped data types in an OCI application:

- Bind and Define Examples
- Salary Update Examples

**See Also:**

- Data Types for a discussion of OCI external data types, and a list of data typecodes
- OCIStringResize()

## 20.16.1 Bind and Define Examples

Shows how you can use variables of type `OCINumber` in OCI bind and define operations

The examples in this section demonstrate how you can use variables of type `OCINumber` 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 that 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;
```

Assume that the `employees` table has been populated with values, and an OCI application has declared a `person` variable:

```
person *my_person;
```

The application then fetches an object into that variable through a `SELECT` statement, such as:

```
text *mystmt = (text *) "SELECT person FROM employees
                        WHERE emp.name='Andrea'";
```

This requires defining `my_person` to be the output variable for this statement, using appropriate OCI define calls for named data types, as described in Advanced Define

Operations in OCI. Executing the statement retrieves the `person` object named `Andrea` into the `my_person` variable.

Once the object is retrieved into `my_person`, the OCI application 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 the following example:

```
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 data type of `VARNUM` (data type 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. The application also defines an output variable for position 1, and uses the appropriate data type 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.

**See Also:**

- Using the Object Type Translator with OCI for a complete discussion of OTT
- OCINumberAssign()

## 20.16.2 Salary Update Examples

Shows how the flow of calls is used to perform certain OCI tasks.

The examples in the previous section demonstrate the flexibility that the Oracle data types provide for bind and define operations. This section shows how you can perform the same operation in several different ways. You can use these data types in variety of ways in OCI applications.

The examples in this section demonstrate the flow of calls used to perform certain OCI tasks. An expanded pseudocode is used for these examples. Actual function names are used, but for simplicity not all parameters and typecasts are filled in. Other necessary OCI calls, such as handle allocations, have been omitted.

**The Scenario**

The scenario for these examples is as follows:

- An employee named *`BRUCE`* exists in the `employees` table for a hospital. See `person` type and `employees` table creation statements in the previous section.
- Bruce's current job title is *`RADIOLOGIST`*.
- Bruce is being promoted to *`RADIOLOGY_CHIEF`*, and along with the promotion comes a salary increase.
- Hospital salaries are in whole dollar values, are set according to job title, and are stored in a table called `salaries`, defined as follows:

  ```
  CREATE TABLE salaries
  (job_title   varchar2(20),
  salary       integer));
  ```
- Bruce's salary must be updated to reflect his promotion.

To update Bruce's salary to reflect the promotion, the application must 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.

Assume that a variable of type `person` has been declared as follows:

```
person * my_person;
```

The object corresponding to Bruce has been fetched into `person`. The following sections present three different ways in which the salary update could be performed.

This section includes the following topics:

- Method 1 - Fetch, Convert, Assign
- Method 2 - Fetch and Assign
- Method 3 - Direct Fetch
- Summary and Notes

## 20.16.2.1 Method 1 - Fetch, Convert, Assign

Shows how to fetch the new salary, convert the integer data type to `OCINumber`, then assign the new salary to an employee.

Example 20-13 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.

**Example 20-13 Method 1 for a Salary Update: Fetch, Convert, and Assign**

```
#define INT_TYPE 3        /* data type 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 */
```

### 20.16.2.2 Method 2 - Fetch and Assign

Shows how to fetch the salary and get it as an `OCINumber` and then assign the new salary to an employee.

This method (Example 20-14) eliminates one of the steps described 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.

**Example 20-14 Method 2 for a Salary Update: Fetch and Assign, No Convert**

```
#define VARNUM_TYPE 6         /* data type 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 */
```

### 20.16.2.3 Method 3 - Direct Fetch

Shows how to do the entire operation with a single define and fetch.

This method (Example 20-15) 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.

Because the object corresponding to Bruce is pinned in the object cache, use the location of his salary attribute as the define variable, and execute or fetch directly into it.

**Example 20-15 Method 3 for a Salary Update: Direct Fetch**

```
#define VARNUM_TYPE 6         /* data type 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 */
```

### 20.16.2.4 Summary and Notes

Summarizes what the previous examples demonstrated and highlights additional information.

As the previous three examples show, the C data types provide flexibility for binding and defining. In these examples an integer can be fetched, and then converted to an `OCINumber` for manipulation. You can use an `OCINumber` 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 these examples it is important to remember that in OCI, if an output variable is defined before the execution of a query, the resulting data is prefetched directly into the output buffer.

In the preceding examples, extra steps would be necessary to ensure that the application writes changes to the database permanently. These might 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.

## 20.17 SQLT_NTY Bind and Define Examples

The following code fragments demonstrate the use of the `SQLT_NTY` named data type in the bind call including `OCIBindObject()` and the `SQLT_NTY` named data type in the define call including `OCIDefineObject()`.

In each example, a previously defined SQL statement is being processed.

This section includes the following topics:

- SQLT_NTY Bind Example
- SQLT_NTY Define Example

**See Also:**

- OCIBindObject()
- OCIDefineObject()

## 20.17.1 SQLT_NTY Bind Example

Shows how to use the `SQLT_NTY` named data type in the bind call including `OCIBindObject()`.

Example 20-16 shows how to use the `SQLT_NTY` named data type in the bind call including `OCIBindObject()`.

**Example 20-16 Using the SQLT_NTY Bind Call Including OCIBindObject()**

```
/*
** 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, (void  *) &custno,
          (sb4) sizeof(sword), SQLT_INT,
          (void  *) 0, (ub2 *)0, (ub2 *)0, (ub4) 0, (ub4 *) 0,
          (ub4) OCI_DEFAULT));

  checkerr(errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":addr",
          (sb4) -1, (void  *) 0,
          (sb4) 0, SQLT_NTY, (void  *) 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, (void  **) &addr,
        (ub4 *) 0, (void  **) &naddr, (ub4 *) 0));
```

**See Also:**

OCIBindObject()

## 20.17.2 SQLT_NTY Define Example

Shows how to use the `SQLT_NTY` named data type in the define call including `OCIDefineObject()`.

Example 20-17 shows how to use the `SQLT_NTY` named data type in the define call including `OCIDefineObject()`.

**Example 20-17    Using the SQLT_NTY Define Call Including OCIDefineObject()**

```
/*
** This example executes a SELECT statement from a table that 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, (void  *)
       &custno, (sb4) sizeof(sword), SQLT_INT, (void  *) 0, (ub2 *)0,
       (ub2 *)0, (ub4) OCI_DEFAULT));

checkerr(errhp, OCIDefineByPos(stmthp, &defn2p, errhp, (ub4) 2, (void  *)
        0, (sb4) 0, SQLT_NTY, (void  *) 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, (void  **)
       &addr, (ub4 *) 0, (void  **) 0, (ub4 *) 0));

  checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
       (OCISnapshot *) NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));
```

**See Also:**

OCIDefineObject()

# 21

# Direct Path Load Interface

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

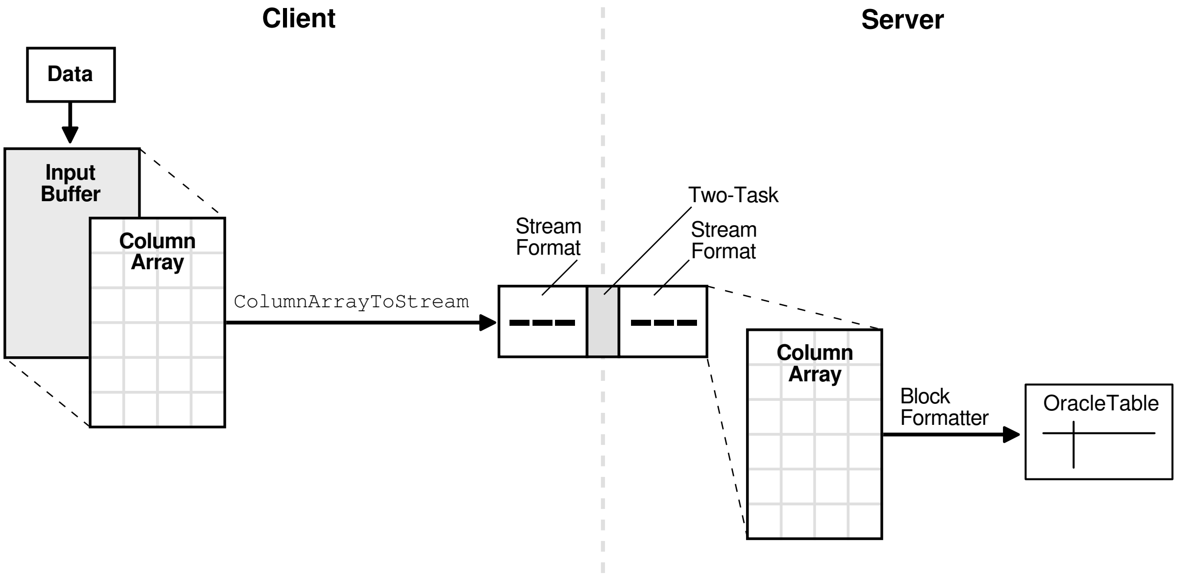
## 21.1 Direct Path Loading Overview

The direct path load interface enables an OCI application to access the direct path load engine of Oracle Database to perform the functions of the 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 21-1 introduces the subject of this chapter. On the client side of the illustration, data enters a column array through an input buffer. The `OCIDirPathColArrayToStream()` call moves the data to the server side through stream formats. These pass data to a column array that uses a block formatter to send the data to the database table.

**Figure 21-1 Direct Path Loading**


Client
Server
Data
Input Buffer
Column Array
ColumnArrayToStream
Stream Format
Two-Task
Stream Format
Column Array
Block Formatter
OracleTable

The OCI direct path load interface can 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. Initialize OCI.
2. Allocate a direct path context handle and set the attributes.
3. Supply the name of the object (table, partition, or subpartition) to be loaded.
4. Describe the external data types of the columns of the object.
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 operations 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 subpartition load is being done as opposed to an entire table load.

- 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 are acquired
  - TRUE, then the table DML SS-Lock and partition DML S-Lock are acquired

This section includes the following topics:

- Data Types Supported for Direct Path Loading
- Direct Path Handles
- About Direct Path Interface Functions
- Limitations and Restrictions of the Direct Path Load Interface
- Direct Path Load Examples for Scalar Columns
- About Using a Date Cache in Direct Path Loading of Dates in OCI
- About Validating Format for Oracle NUMBER and DATE Data

**See Also:**

- Direct Path Context Handle (OCIDirPathCtx) Attributes
- OCIDirPathColArrayToStream()

## 21.1.1 Data Types Supported for Direct Path Loading

Lists the external data types that are valid for scalar columns in a direct path load operation.

The following external data types 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_CLOB`
- `SQLT_BLOB`
- `SQLT_DATE`
- `SQLT_TIMESTAMP`
- `SQLT_TIMESTAMP_TZ`
- `SQLT_TIMESTAMP_LTZ`
- `SQLT_INTERVAL_YM`
- `SQLT_INTERVAL_DS`

The following external object data types 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:**
>
> - About Accessing Column Parameter Attributes for information on setting or retrieving the data type of a column
> - Table 5-2 for information about data types

## 21.1.2 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
- OCI Direct Path Function Context
- Direct Path Column Array and Direct Path Function Column Array
- Direct Path Stream

> **See Also:**
>
> Direct Path Loading Handle Attributes and all the descriptions of direct path attributes that follow

### 21.1.2.1 Direct Path Context

The direct path context handle must be allocated for each object, either a table or a partition of a partitioned table, being loaded.

Because an `OCIDirPathCtx` handle is the parent handle of the `OCIDirPathFuncCtx`, `OCIDirPathColArray`, and `OCIDirPathStream` handles, freeing an `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, as shown in Example 21-1.

**Example 21-1 Direct Path Programs Must Include the Header Files**

```
...
#include <cdemodp0.h>
#include <cdemodp.h>

OCIEnv *envp;
OCIDirPathCtx *dpctx;
sword error;
error = OCIHandleAlloc((void  *)envp, (void  **)&dpctx,
                    OCI_HTYPE_DIRPATH_CTX, (size_t)0,(void  **)0);
```

```
...
error = OCIHandleFree(dpctx, OCI_HTYPE_DIRPATH_CTX);
```

**See Also:**

- OCIHandleAlloc()
- OCIHandleFree()

### 21.1.2.2 OCI Direct Path Function Context

Use the direct path function context handle to describe named type and `REF` columns: column objects, REF columns, and SQL string columns.

The direct path function context 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 is to 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 each 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 shown in Example 21-2.

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 `OCIHandleFree()`:

```
error = OCIHandleFree(dpfnctx, OCI_HTYPE_DIRPATH_FN_CTX);
```

**Example 21-2    Passing the Handle Type to Allocate the Function Context**

```
OCIDirPathCtx *dpctx;       /* direct path context */
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
sword error;

error = OCIHandleAlloc((void  *)dpctx, (void  **)&dpfnctx,
                       OCI_HTYPE_DIRPATH_FN_CTX,
                       (size_t)0, (void  **)0);
```

**See Also:**

- *Oracle Database Object-Relational Developer's Guide* for more information about the data types supported
- OCIHandleAlloc()
- OCIHandleFree()

## 21.1.2.3 Direct Path Column Array and Direct Path Function Column Array

The direct path column array handle and direct path function column handle 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 code fragment in Example 21-3 shows explicit allocation of the direct path column array handle.

A direct path column array handle is freed with `OCIHandleFree()`.

```
error = OCIHandleFree(dpca, OCI_HTYPE_DIRPATH_COLUMN_ARRAY);
```

Example 21-4 shows that a direct path function column array handle is allocated in almost the same way.

A direct path function column array is freed with `OCIHandleFree()`:

```
error = OCIHandleFree(dpfnca, OCI_HTYPE_DIRPATH_FN_COL_ARRAY);
```

Freeing an `OCIDirPathColArray` handle also frees the column array associated with the handle.

**Example 21-3    Explicit Allocation of Direct Path Column Array Handle**

```
OCIDirPathCtx *dpctx;      /* direct path context */
OCIDirPathColArray *dpca;  /* direct path column array */
sword error;
error = OCIHandleAlloc((void  *)dpctx, (void  **)&dpca,
               OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
               (size_t)0, (void  **)0);
```

**Example 21-4    Explicit Allocation of Direct Path Function Column Array Handle**

```
OCIDirPathFuncCtx *dpfnctx;   /* direct path function context */
OCIDirPathColArray *dpfnca;   /* direct path function column array */
sword error;
error = OCIHandleAlloc((void  *)dpfnctx, (void  **)&dpfnca,
                        (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
                        (size_t)0, (void  **)0);
```

**See Also:**

OCIHandleAlloc()

## 21.1.2.4 Direct Path Stream

A direct path stream is a linear representation of Oracle table data.

The direct path stream handle is used by the conversion operation, `OCIDirPathColArrayToStream()`, and by the load operation, `OCIDirPathLoadStream()`.

Direct path stream handles are allocated by the client with `OCIHandleAlloc()`. The structure of an `OCIDirPathStream` handle can be thought of as a pair in the form (buffer, buffer length).

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()`.

Example 21-5 shows a direct path stream handle allocated with `OCIHandleAlloc()`. The parent handle is always an `OCIDirPathCtx` handle.

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.

**Example 21-5    Allocating a Direct Path Stream Handle**

```
OCIDirPathCtx *dpctx;    /* direct path context */
OCIDirPathStream *dpstr; /* direct path stream */
sword error;
error = OCIHandleAlloc((void  *)dpctx, (void  **)&dpstr,
               OCI_HTYPE_DIRPATH_STREAM, (size_t)0,(void  **)0);
```

**See Also:**

- OCIDirPathColArrayToStream()
- OCIDirPathLoadStream()
- OCIHandleAlloc()
- OCIDirPathStreamReset()

## 21.1.3 About Direct Path Interface Functions

Lists and describes the functions used with the direct path load interface.

The functions listed in this section are used with the direct path load interface.

Operations on the direct path context are performed by the functions in Table 21-1.

**Table 21-1    Direct Path Context Functions**

| Function | Purpose |
|---|---|
| OCIDirPathAbort() | Terminates a direct path operation |
| OCIDirPathDataSave() | Executes a data savepoint |
| OCIDirPathFinish() | Commits the loaded data |

**Table 21-1 (Cont.) Direct Path Context Functions**

| Function | Purpose |
|---|---|
| OCIDirPathFlushRow() | Flushes a partially loaded row from the server. This function is deprecated. |
| OCIDirPathLoadStream() | Loads the data that has been converted to direct path stream format |
| OCIDirPathPrepare() | Prepares the direct path interface to convert or load rows |

Operations on the direct path column array are performed by the functions in Table 21-2.

**Table 21-2 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() that resets the direct stream state.

**See Also:**

Direct Path Loading Functions for detailed descriptions of each function

## 21.1.4 Limitations and Restrictions of the Direct Path Load Interface

Lists the limitations 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 you must determine what partition the LOB goes into before you start writing to it.

## 21.1.5 Direct Path Load Examples for Scalar Columns

This section describes the direct path load examples for scalar columns.

This is described in the following topics:

- Data Structures Used in Direct Path Loading Example
- Outline of an Example of a Direct Path Load for Scalar Columns

### 21.1.5.1 Data Structures Used in Direct Path Loading Example

Shows examples that use the data structures in direct path loading.

Example 21-6 shows the data structure used in Example 21-7 through Example 21-17.

Example 21-7 shows the header file `cdemodp.h` from the `demo` directory, which defines several structs.

**Example 21-6    Data Structures Used in Direct Path Loading Examples**

```
/* 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 */
};
```

**Example 21-7    Contents of the Header File cdemodp.h**

```
#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 */
```

## 21.1.5.2 Outline of an Example of a Direct Path Load for Scalar Columns

Shows sample code that illustrates the use of several of the OCI direct path interfaces.

Example 21-8 shows sample code that 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.

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, as shown in Example 21-9. Also note that errors are returned with the environment handle associated with the direct path context.

**Get the Number of Rows and Columns**

Get the number of rows and columns in the column array just allocated, as shown in Example 21-10.

**Set the Input Data Fields**

Set the input data fields to their corresponding data columns, as shown in Example 21-11.

**Reset the Column Array State**

Reset the column array state in case a previous conversion must be continued or a row is expecting more data, as shown in Example 21-12.

**Reset the Stream State**

Reset the stream state to start a new stream, as shown in Example 21-13. Otherwise, data in the stream is appended to existing data.

**Convert the Data in the Column Array to Stream Format**

After inputting the data, convert the data in the column array to stream format and filter out any bad records, as shown in Example 21-14.

### 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 that produced the stream. When the conversion to stream format is done, the data is appended to the stream, as shown in Example 21-15. It is the responsibility of the caller to reset the stream when appropriate. On errors, the position is moved to the next row, or to 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 an `OCIDirPathLoadStream()` call is made and the end of a stream has been reached, `OCI_NO_DATA` is returned.

### Finish the Direct Path Load

Finish the direct path load as shown in Example 21-16.

### Free the Direct Path Handles

Free all the direct path handles allocated, as shown in Example 21-17. Note that the direct path column array and stream handles are freed before the parent direct path context handle is freed.

**Example 21-8    Use of OCI Direct Path Interfaces**

```
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((void  *)ctlp->envhp_ctl,
                           (void  **)&ctlp->dpctx_ctl,
                           (ub4)OCI_HTYPE_DIRPATH_CTX,
                           (size_t)0, (void  **)0));

  dpctx = ctlp->dpctx_ctl;                                      /* shorthand */

  OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
            OCIAttrSet((void  *)dpctx, (ub4)OCI_HTYPE_DIRPATH_CTX,
                       (void  *)tblp->name_tbl,
                       (ub4)strlen((const char *)tblp->name_tbl),
                       (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl));
```

**Example 21-9    Allocating the Column Array and Stream Handles**

```
   OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
          OCIHandleAlloc((void  *)ctlp->dpctx_ctl, (void  **)&ctlp->dpca_ctl,
                         (ub4)OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                         (size_t)0, (void  **)0));

OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
          OCIHandleAlloc((void  *)ctlp->dpctx_ctl,(void  **)&ctlp->dpstr_ctl,
                         (ub4)OCI_HTYPE_DIRPATH_STREAM,
                         (size_t)0, (void  **)0));
```

### Example 21-10 Getting the Number of Rows and Columns

```
  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));
```

### Example 21-11 Setting Input Data Fields

```
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));
```

### Example 21-12 Resetting the Column Array State

```
(void) OCIDirPathColArrayReset(ctlp->dpca_ctl, ctlp->errhp_ctl);
```

### Example 21-13 Resetting the Stream State

```
(void) OCIDirPathStreamReset(ctlp->dpstr_ctl, ctlp->errhp_ctl);
```

### Example 21-14 Converting Data to Stream Format

```
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);
```

### Example 21-15 Loading the Stream

```
/* load the stream */
ociret = OCIDirPathLoadStream(ctlp->dpctx_ctl, ctlp->dpstr_ctl,
            ctlp->errhp_ctl);
```

**Example 21-16 Finishing the Direct Path Load Operation**

```
/* finish the direct path load operation */
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIDirPathFinish(ctlp->dpctx_ctl, ctlp->errhp_ctl));
```

**Example 21-17 Freeing the Direct Path Handles**

```
/* free up server data structures for the load */
ociret = OCIHandleFree((void  *)ctlp->dpca_ctl,
                           OCI_HTYPE_DIRPATH_COLUMN_ARRAY);
ociret = OCIHandleFree((void  *)ctlp->dpstr_ctl,
                           OCI_HTYPE_DIRPATH_STREAM);
ociret = OCIHandleFree((void  *)ctlp->dpctx_ctl,
                          OCI_HTYPE_DIRPATH_CTX);
```

**See Also:**

OCIDirPathLoadStream()

## 21.1.6 About Using a Date Cache in Direct Path Loading of Dates in OCI

The *date cache* feature provides improved performance when loading Oracle date and time-stamp values that require data type conversions to be stored in the table.

For more information about using this feature in direct path loading, see *Oracle Database Utilities*.

This feature is specifically targeted to direct path loads where the same input date or timestamp values are being loaded over and over again. Date conversions are very expensive and can account for a large percentage of the total load time, especially if multiple date columns are being loaded. The date cache 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 only improves performance when many duplicate input date values are being loaded into date columns (the word *date* in this chapter applies to all the date and time-stamp data types).

The date cache is enabled by default. When you explicitly specify the date cache size, the date cache feature is not disabled, by default. To override this behavior, set `OCI_ATTR_DIRPATH_DCACHE_DISABLE` to 1. Otherwise, the cache continues to be searched to avoid date conversions. However, any misses (entries for which there are no duplicate values) are converted the usual way using expensive date conversion functions without the benefit of using the date cache feature.

Query the attributes `OCI_ATTR_DIRPATH_DCACHE_NUM`, `OCI_ATTR_DIRPATH_DCACHE_MISSES`, and `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 (entries for which there are duplicate values). Excessive date cache misses, however, can cause the application to run slower than not using the date cache at all. 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
- OCI_ATTR_DIRPATH_DCACHE_NUM
- OCI_ATTR_DIRPATH_DCACHE_MISSES
- OCI_ATTR_DIRPATH_DCACHE_HITS
- OCI_ATTR_DIRPATH_DCACHE_DISABLE

### 21.1.6.1 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 time-stamp 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 is not created for a table. A date cache is created for a table only if one or more date or time-stamp values are loaded that require data type conversions and the attribute value is nonzero.

### 21.1.6.2 OCI_ATTR_DIRPATH_DCACHE_NUM

This attribute is used to query the current number of entries in a date cache.

### 21.1.6.3 OCI_ATTR_DIRPATH_DCACHE_MISSES

This attribute is used to query the current number of date cache misses. If the number of misses is high, consider using a larger date cache size. If increasing the date cache size does not cause this number to decrease significantly, the date cache should probably not be used. Date cache misses are expensive, due to hashing and lookup times.

### 21.1.6.4 OCI_ATTR_DIRPATH_DCACHE_HITS

This attribute is used to query the number of date cache hits. This number should be relatively large to see any benefit of using the date cache support.

### 21.1.6.5 OCI_ATTR_DIRPATH_DCACHE_DISABLE

Used to disable the data cache if the size is exceeded.

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 are not added to the date cache, and are 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
- OCIDirPathPrepare()

### 21.1.7 About Validating Format for Oracle NUMBER and DATE Data

Describes validating format for Oracle NUMBER and DATE data when loading stream data.

Use the direct path context handle attribute `OCI_ATTR_DIRPATH_FLAGS` with the `OCI_DIRPATH_FLAGS_VLDT` 0x01 flag set to validate format for Oracle `NUMBER` and `DATE` data when the stream is parsed on the server. The default value is to not set this flag because it is an expensive operation. It can be used when you suspect that there is a problem with `OCIDirPath` generating invalid internal representation of dates and numbers.

See Direct Path Context Handle (OCIDirPathCtx) Attributes for more information.

## 21.2 Direct Path Loading of Object Types

The use of the direct path function contexts to load various nonscalar types is discussed in this section.

The nonscalar 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

This section includes the following topics:

- Direct Path Loading of Nested Tables
- Direct Path Loading of Column Objects
- Direct Path Loading of SQL String Columns
- Direct Path Loading of REF Columns
- Direct Path Loading of NOT FINAL Object and REF Columns
- Direct Path Loading of Object Tables
- Direct Path Loading a NOT FINAL Object Table

> **See Also:**
>
> Table B-1 for a listing of the programs demonstrating direct path loading that are available with your Oracle Database installation

## 21.2.1 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 for orphaned children to be created when the rows are inserted in to the child table, but the corresponding parent row is not inserted in to the parent table. It is also possible to insert a parent row in to the parent table without inserting child rows in to the child table, so that the parent row has missing children.

### 21.2.1.1 Describing a Nested Table Column and Its Nested Table

Lists and describes loading the parent table with a nested table column, whichis a separate action from loading the child nested table.

> **Note:**
>
> Steps that are different from loading scalar data are in italic.

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 data type is SQLT_CHR if the SETIDs in the data file are in characters, SQLT_BIN if binary.*
- *To load the nested table (child):*
  1. Describe the nested table and its columns as usual.
  2. *The SETID column is required.*

  - *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((void  *)colDesc,
                       OCI_DTYPE_PARAM,
                       (void  *)&flg, (ub4)0,
                       OCI_ATTR_DIRPATH_SID, ctlp->errhp_ctl);
    ```

This section includes the following topic: Describing a Nested Table Column and Its Nested Table.

## 21.2.2 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.

This section includes the following topics:

- Describing a Column Object
- Allocating the Array Column for the Column Object
- Loading Column Object Data into the Column Array
- OCI_DIRPATH_COL_ERROR

### 21.2.2.1 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 shown here are similar to those describing a list of scalar columns to be loaded for a table. Steps that are different from loading scalar data are in italic.

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 is used to describe the column's object type and attributes:*

   ```
   OCIDirPathFuncCtx *dpfnctx  /* direct path function context */;
   sword error;
   error = OCIHandleAlloc((void  *)dpctx, (void  **)&dpfnctx,
                  OCI_HTYPE_DIRPATH_FN_CTX,
                  (size_t)0, (void  **)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((void  *)dpfnctx,
                      OCI_HTYPE_DIRPATH_FN_CTX,
                      (void  *)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 is used as the default object constructor:*

   ```
   OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
   ub1 expr_type = OCI_DIRPATH_EXPR_OBJ_CONSTR;
   sword error;

   error = OCIAttrSet((void  *)dpfnctx,
                      OCI_HTYPE_DIRPATH_FN_CTX,
                      (void  *)&expr_type, (ub4)0,
                      OCI_ATTR_DIRPATH_EXPR_TYPE,
                      ctlp->errhp_ctl);
   ```

7. *Set the number of columns or object attributes that are to be loaded for this column object using OCI_ATTR_NUM_COLS.*
8. *Get the column or attribute parameter list for the function context `OCIDirPathFuncCtx`.*
9. *For each object attribute:*
   a. Get the column descriptor for the object attribute with `OCI_DTYPE_PARAM`.
   b. Set the attribute's column name with `OCI_ATTR_NAME`.
   c. Set the external column type (the type of the data that is to be passed to the direct path API) with `OCI_ATTR_DATA_TYPE`.
   d. Set any other external column attributes (maximum data size, precision, scale, and so on.)
   e. *If this attribute column is a column object, then do Steps 3 through 10 for its object attributes.*
   f. 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.*

### 21.2.2.2 Allocating the Array Column for the Column Object

When you direct path load a column object, the data for its object attributes is 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 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 function column array value `OCI_HTYPE_DIRPATH_FN_COL_ARRAY`.

Example 21-18 shows how to allocate a child column array for a column object.

**Example 21-18    Allocating 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((void  *)dpfnctx, (void  **)&dpfnca,
                       OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
                       (size_t)0, (void  **)0);
```

### 21.2.2.3 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()`.

If a column is an object, the address of its child column array handle is passed instead. The child column array contains the data of the object attributes.

To load data into a column object:

> **Note:**
>
> Steps that are different from loading scalar data are in italic.

*(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()`.

**See Also:**

OCIDirPathColArrayEntrySet()

### 21.2.2.4 OCI_DIRPATH_COL_ERROR

The `OCI_DIRPATH_COL_ERROR` 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 that 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 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.

**See Also:**

OCIDirPathColArrayToStream()

## 21.2.3 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, sequences, and `LONG`s.

A SQL expression is represented to the direct path API using the `OCIDirPathFuncCtx`. Its `OCI_ATTR_NAME` value is 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 are 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.

This section includes the following topics:

- Describing a SQL String Column
- Allocating the Column Array for SQL String Columns
- Loading the SQL String Data into the Column Array

## 21.2.3.1 Describing a SQL String Column

Lists steps for describing a SQL string column.

> **Note:**
>
> Steps that are different from loading scalar data are in italic.

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 is used to describe the arguments of the SQL string.*

   ```
   OCIDirPathFuncCtx *dpfnctx  /* direct path function context */;
   sword error;
   error = OCIHandleAlloc((void  *)dpctx, (void  **)&dpfnctx,
                  OCI_HTYPE_DIRPATH_FN_CTX,
                  (size_t)0, (void  **)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((void  *)dpfnctx,
                      OCI_HTYPE_DIRPATH_FN_CTX,
                      (void  *)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 is 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((void  *)dpfnctx,
                      OCI_HTYPE_DIRPATH_FN_CTX,
   ```

```
                       (void  *)&expr_type, (ub4)0,
                       OCI_ATTR_DIRPATH_EXPR_TYPE, ctlp->errhp_ctl);
```

7. *Set the number of arguments that are to be passed to the SQL string with OCI_ATTR_NUM_COLS.*
8. *Get the column or attribute parameter list for the function context.*
9. *For each SQL string argument:*
    1. Get the column descriptor for the object attribute with `OCI_DTYPE_PARAM`.
    2. *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.*
    3. Set the attribute's column name with `OCI_ATTR_NAME`.
    4. *Use the naming convention for SQL string arguments.*
    5. *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".*
    6. *If an argument is used multiple times in a SQL string, declaring it once and counting it as one argument only is correct.*
    7. Set the external column type (the type of the data that is to be passed to the direct path API) with `OCI_ATTR_DATA_TYPE`.
    8. Set any other external column attributes (maximum data size, precision, scale, and so on).
    9. 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.*

## 21.2.3.2 Allocating the Column Array for SQL String Columns

When you direct path load a SQL string column, the data for its arguments is 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 to the corresponding non-NULL row of its parent SQL string column in the parent column array.

Example 21-19 shows how to allocate a child column array for a SQL string column.

**Example 21-19    Allocating 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((void  *)dpfnctx, (void  **)&dpfnca,
                       OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
                       (size_t)0, (void  **)0);
```

### 21.2.3.3 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 italic.

*For each SQL string column:*

1. *If the column is non-NULL:*
    1. *For each of its function argument columns:*

       *Set the data in the child column array using OCIDirPathColArrayEntrySet().*
    2. *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 for more information about passing the `OCI_DIRPATH_COL_ERROR` value to `OCIDirPathColArrayEntry()` to indicate that the current column array row should be ignored when an error occurs
> - OCIDirPathColArrayEntrySet()

## 21.2.4 Direct Path Loading of REF Columns

The `REF` type is a pointer, or reference, to a row object in an object table.

This section includes the following topics:

- Describing the REF Column
- Allocating the Column Array for a REF Column

- Loading the REF Data into the Column Array

### 21.2.4.1 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 italic.

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((void  *)dpctx, (void  **)&dpfnctx,
                          OCI_HTYPE_DIRPATH_FN_CTX,
                          (size_t)0, (void  **)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((void  *)dpfnctx,
                      OCI_HTYPE_DIRPATH_FN_CTX,
                      (void  *)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 is to 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 shortcut 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 uses 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 must match the table name specified in the schema. If it is not in the data row, the API uses the reference table specified in the schema.*

7. *Set the number of REF arguments that are to 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 is 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 pass in some `REF` arguments other than N or N+1, the error message says 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 does not invoke an error message.

8. *Get the column or 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. Because 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 shortcut note later), can be given in any order.*

*Do not set the attribute column names for the object ID to use the shortcut.*

***Shortcut.*** *If loading a system-OID-based REF column, do not set the column name with a name. The API figures it out. But you must still set other column attributes, such as external data type.*

*If loading a primary-key REF column and its primary key consists of multiple columns, the shortcut is not to set their column names. But you must still set other column attributes, such as external data type.*

> **Note:**
>
> If the component column names are NULL, then the API code determines the column names in the position or order in which they were defined for the primary key. So, when you set column attributes other than the name, ensure that the attributes are set for the component columns in the correct order.

c. Set the external column type (the type of the data that is to 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.*

## 21.2.4.2 Allocating the Column Array for a REF Column

Shows how to allocate a child column array for a `REF` column.

Example 21-20 shows how to allocate a child column array for a `REF` column.

**Example 21-20 Allocating 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((void  *)dpfnctx, (void  **)&dpfnca,
                       OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
                       (size_t)0, (void  **)0);
```

### 21.2.4.3 Loading the REF 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()`.

If a column is a `REF`, the address of its child column array handle is passed instead. The child column array contains the `REF` arguments' data.

To load data into a `REF` column:

> **Note:**
>
> Steps that are different from loading scalar data are in italic.

*For each REF column:*

1. *If the column is non-NULL:*
   1. *For each of its REF argument columns:*

      *Set its data in the child column array using OCIDirPathColArrayEntrySet().*
   2. *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, a length of 0, and an `OCI_DIRPATH_COL_NULL` flag to `OCIDirPathColArrayEntrySet()`.

> **See Also:**
>
> - OCI_DIRPATH_COL_ERROR
> - OCIDirPathColArrayEntrySet()

## 21.2.5 Direct Path Loading of NOT FINAL Object and REF Columns

Recall that SQL object inheritance is based on a family tree of object types that forms a type hierarchy.

The type hierarchy consists of a parent object type, called a supertype, and one or more levels of child object types, called subtypes, which are derived from the parent.

This section includes the following topics:

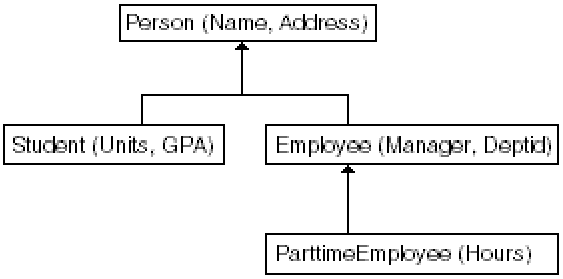
- Inheritance Hierarchy
- About Describing a Fixed, Derived Type to Be Loaded
- About Allocating the Column Array
- About Loading the Data into the Column Array

### 21.2.5.1 Inheritance Hierarchy

Recall that for an object type to be inheritable, the object type definition must specify that it is inheritable.

Figure 21-2 diagrams the inheritance hierarchy for a column of type `Person`. The `Person` supertype is at the top of the hierarchy with two attributes: `Name`, `Address`. `Person` has two subtypes, `Employee` and `Student`. The `Employee` subtype has two attributes: `Manager`, `Deptid`. The `Student` subtype has two attributes: `Units`, `GPA`. `ParttimeEmployee` is a subtype of `Employee` and appears below it. The subtype `ParttimeEmployee` has one attribute: `Hours`. These are the types that can be stored in a `Person` column.

**Figure 21-2 Inheritance Hierarchy for a Column of Type Person**



Once specified, subtypes can be derived from it. To specify an object to be inheritable, the keyword `NOT FINAL` must be specified in its type definition. To specify an object to not be inheritable, the keyword FINAL must be specified in its type definition.

When you direct path load a table that 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 subtypes: `Student`, `Employee`, and `ParttimeEmployee`. Because the direct path load API only supports the loading of one fixed, derived type to this `NOT FINAL` column for the duration of this load, the direct path load API must know which one of these types is to be loaded, the attributes to load for this type, and the function used to create this type.

So when you describe and load a derived type, you must specify all of the attributes for that type that are to be loaded. 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, you must describe and count.

For example, to load all columns in `ParttimeEmployee`, you must describe and count five object attributes to load into: `Name`, `Address`, `Manager`, `Deptid`, and `Hours`.

**See Also:**

*Oracle Database Object-Relational Developer's Guide* for more information about defining `FINAL` and `NOT FINAL` types.

### 21.2.5.2 About Describing a Fixed, Derived Type to Be Loaded

Note that the steps to describe a `NOT FINAL` or substitutable object columns and `REF` columns of a fixed, derived type are similar to the steps that describe its `FINAL` counterpart.

These sections 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 describing 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 must be described and counted.

> **See Also:**
>
> Direct Path Loading of Column Objects or Direct Path Loading of REF Columns for information about how to describe a `NOT FINAL` column of type X (where X is an object or a `REF`)

### 21.2.5.3 About Allocating the Column Array

Indicates it is the same as for a `FINAL` column of the same type.

Allocating the column array is the same as for a `FINAL` column of the same type.

### 21.2.5.4 About Loading the Data into the Column Array

Indicates it is the same as for a `FINAL` column of the same type.

Loading the data into the column array is the same as for a `FINAL` column of the same type.

## 21.2.6 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 italic.

*For each object attribute column:*

Describe each object attribute column as it must 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:*

   *There is nothing extra to do. The system generates 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. *Load all of the primary-key columns making up the OID.*
   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**

Example 21-21 shows that allocating the column array for the object table is the same as allocating a column array for a non-object table.

**Loading Data into the Column Array**

Loading data into the column array is the same as loading data into a non-object table.

**Example 21-21 Allocating the Column Array for the Object Table**

```
OCIDirPathColArray *dpca;  /* direct path column array */
sword error;

error = OCIHandleAlloc((void  *)dpctx, (void  **)&dpca,
                       OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                       (size_t)0, (void  **)0);
```

## 21.2.7 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 describing a `FINAL` object table are in italic.

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, are 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((void  *)dpctx,
                   OCI_HTYPE_DIRPATH_CTX,
                   (void  *) obj_type,
                   (ub4)strlen((const char *) obj_type),
                   OCI_ATTR_DIRPATH_OBJ_CONSTR, ctlp->errhp_ctl);
```

2. Describe according to its data type each of the object attribute columns to be loaded. 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**

Allocating the column array for the `NOT FINAL` object table is the same as for a `FINAL` object table.

# 21.3 Direct Path Loading in Pieces

To support loading data that does not all fit in memory at one time, use loading in pieces.

The direct path API supports loading `LONG`s and LOBs incrementally. This is accomplished through the following steps:

1. Set the first piece into the column array using `OCIDirPathColArrayEntrySet()` and passing in the `OCI_DIRPATH_COL_PARTIAL` flag to indicate that all the data for this column has not been loaded yet.
2. Convert the column array to a stream.
3. Load the stream.
4. Set the next piece of that data into the column array. If it is not complete, set the partial flag and go back to Step 2. If it is complete, then set the `OCI_DIRPATH_COL_COMPLETE` flag and continue to the next column.

This approach is essentially the same for dealing with large attributes for column objects and large arguments for SQL string types.

> **Note:**
>
> Collections are not loaded in pieces, as such. Nested tables are loaded separately and are loaded like a top-level table. Nested tables can be loaded incrementally and can have columns that are loaded in pieces. Therefore, do not set the `OCI_DIRPATH_COL_PARTIAL` flag for the column containing the collection.
>
> This section includes the following topic: Loading Object Types in Pieces.

**See Also:**

- OCI_DIRPATH_COL_ERROR for more information about passing the `OCI_DIRPATH_COL_ERROR` value to `OCIDirPathColArrayEntry()` to indicate that the current column array row should be ignored when an error occurs
- OCIDirPathColArrayEntrySet()

## 21.3.1 Loading Object Types in Pieces

Objects are loaded into a separate column array from the parent table that 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. Convert the parent column array to a stream. This converts the child column array as well.
4. Load the stream.
5. Go back to Step 1 and continue loading the remaining data for that element until it is complete.

Here are some rules about loading in pieces:

- There can only be one partial element at a time at any level. Once one partial element is marked complete, then another one at that level can 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 is a top-level table.

**See Also:**

OCI_DIRPATH_COL_ERROR for more information about passing the `OCI_DIRPATH_COL_ERROR` value to `OCIDirPathColArrayEntry()` to indicate that the current column array row should be ignored when an error occurs.

# 21.4 Direct Path Context Handles and Attributes for Object Types

The following discussion gives the supplemental details of the handles and attributes.

This section includes the following topics:

- Direct Path Context Attributes
- Direct Path Function Context and Attributes
- Direct Path Column Parameter Attributes
- Direct Path Function Column Array Handle for Nonscalar Columns

> **See Also:**
>
> Handle and Descriptor Attributes for a discussion that gives the supplemental details of the handles and attributes

## 21.4.1 Direct Path Context Attributes

There is one direct path context attribute.

> **See Also:**
>
> OCI_ATTR_DIRPATH_OBJ_CONSTR is the only direct path context attribute

### 21.4.1.1 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((void  *)dpctx,
                   OCI_HTYPE_DIRPATH_CTX,
                   (void  *) obj_type,
                   (ub4)strlen((const char *) obj_type),
                   OCI_ATTR_DIRPATH_OBJ_CONSTR, ctlp->errhp_ctl);
```

## 21.4.2 Direct Path Function Context and Attributes

Here is a summary of the attributes for function context handles.

This section includes the following topics:

- OCI_ATTR_DIRPATH_OBJ_CONSTR

- OCI_ATTR_NAME
- OCI_ATTR_DIRPATH_EXPR_TYPE
- OCI_ATTR_DIRPATH_NO_INDEX_ERRORS
- OCI_ATTR_NUM_COLS
- OCI_ATTR_NUM_ROWS

> **See Also:**
>
> Direct Path Context Handle (OCIDirPathCtx) Attributes

## 21.4.2.1 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((void  *)dpctx,
                   OCI_HTYPE_DIRPATH_CTX,
                   (void  *) obj_type,
                   (ub4)strlen((const char *) obj_type),
                   OCI_ATTR_DIRPATH_OBJ_CONSTR, ctlp->errhp_ctl);
```

## 21.4.2.2 OCI_ATTR_NAME

When a function context is created, set `OCI_ATTR_NAME` equal to the expression that describes the nonscalar column.

Then set an OCI attribute to indicate the type of the expression. The expression type varies depending on whether it is a column object, a `REF` column, or a SQL string column.

**Column Objects**

This required expression is the object type name. The object type is used as the default object constructor.

Set the expression type `OCI_ATTR_DIRPATH_EXPR_TYPE` to `OCI_DIRPATH_EXPR_OBJ_CONSTR` to indicate that this expression is an object type name.

**REF Columns**

This optional expression is the reference table name. This table is the object table from which the `REF` column is to reference row objects.

Set the expression type `OCI_ATTR_DIRPATH_EXPR_TYPE` to `OCI_DIRPATH_EXPR_REF_TBLNAME` to indicate that this expression is a reference object table.

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 must 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. The `REF` column is not allowed to have a reference table name as its argument. (Direct path API provides this parameter as a shortcut for 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 uses 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 must match the table name specified in the schema. If it is not in the data row, the API uses the reference table defined in the schema.

**SQL String Columns**

This mandatory expression contains a SQL string to derive the value that is to 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.

### 21.4.2.3 OCI_ATTR_DIRPATH_EXPR_TYPE

This attribute is used to indicate the type of the expression specified in `OCI_ATTR_NAME` for the nonscalar 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:

- `OCI_DIRPATH_EXPR_OBJ_CONSTR`
  - Indicates that the expression is an object type name and is to be used as the default object constructor for a column object.
  - Is required for column objects.
- `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 is referencing row objects.
  - Is optional for `REF` columns.
- `OCI_DIRPATH_EXPR_SQL`
  - Indicates that the expression is a SQL string that is executed to derive a value to be stored in the column.
  - Is required for SQL string columns.

Example 21-22 shows the pseudocode that illustrates the preceding rules and values.

**Example 21-22 Specifying Values for the OCI_ATTR_DIRPATH_EXPR_TYPE Attribute**

```
OCIDirPathFuncCtx  *dpfnctx; /* function context for this nonscalar 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((void  *)(dpfnctx),
                   OCI_HTYPE_DIRPATH_FN_CTX,
                   (void  *)&expr_type, (ub4)0,
                   OCI_ATTR_DIRPATH_EXPR_TYPE, ctlp->errhp_ctl);
```

## 21.4.2.4 OCI_ATTR_DIRPATH_NO_INDEX_ERRORS

When `OCI_ATTR_DIRPATH_NO_INDEX_ERRORS` is 1, indexes are not set unusable at any time during the load.

If any index errors are detected, the load is terminated. That is, no rows are loaded, and the indexes are left as is. The default is 0.

> **See Also:**
>
> OCI_ATTR_DIRPATH_NO_INDEX_ERRORS

## 21.4.2.5 OCI_ATTR_NUM_COLS

This attribute describes the number of attributes or arguments that are to be loaded or processed for a nonscalar column.

This parameter must be set before the column list can be retrieved. The expression type varies depending on whether it is a column object, a SQL string column, or a `REF` column.

**Column Objects**

The number of object attribute columns to be loaded for this column object.

**SQL String Columns**

The number of arguments to be passed to the SQL string.

If an argument is used multiple times in the function, counting it as one is correct.

**REF Columns**

The number of `REF` arguments to identify the row object the `REF` column should point to.

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 are 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 are 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 1. 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 1 to N.

> **Note:**
>
> To simplify the error message if you pass in some `REF` arguments other than N or N+1, the error message says 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 does not invoke an error message.

## 21.4.2.6 OCI_ATTR_NUM_ROWS

This attribute, when used for an `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 `OCIAttrGet()` is called with OCI_ATTR_NUM_ROWS, the number of rows loaded so far is returned.

However, the attribute `OCI_ATTR_NUM_ROWS`, when used for an `OCI_HTYPE_DIRPATH_CTX` (table-level context), can be both set and 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 derives 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.

**See Also:**

- OCIAttrGet()
- OCIAttrSet()

## 21.4.3 Direct Path Column Parameter Attributes

When you describe 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 the column is a SQL string, then its function context describes the expression to be called. If the column is a REF, its function context describes the reference table name and row object identifiers.

Example 21-23 shows that when you set a function context as a column attribute, `OCI_ATTR_DIRPATH_FN_CTX` is used in the `OCIAttrSet()` call.

Attributes for column parameter context handles follow.

**Example 21-23 Setting a Function Context as a Column Attribute**

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
sword error;

error = OCIAttrSet((void  *)colDesc,
                   OCI_DTYPE_PARAM,
                   (void  *)(dpfnctx), (ub4)0,
                   OCI_ATTR_DIRPATH_FN_CTX, ctlp->errhp_ctl);
```

This section includes the following topics:

- OCI_ATTR_NAME
- OCI_ATTR_DIRPATH_SID
- OCI_ATTR_DIRPATH_OID

**See Also:**

- Direct Path Column Parameter Attributes
- OCIAttrSet()

### 21.4.3.1 OCI_ATTR_NAME

The naming conventions for 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 you 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 does not 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 must be set so that the function can identify the column even though it is not under the name known to the database.

The naming rules are as follows:

- Child nested table's `SETID` (`SID`) column

  The SETID column is required. Set its `OCI_ATTR_NAME` using a dummy name, because the API does not 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.

- Object table's object ID (`OID`) column

  An object ID is required if:

  1. 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`.

  2. 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`.

- SQL string argument

  1. Set the attribute's column name with `OCI_ATTR_NAME`.
  2. 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.
  3. Use the naming convention for SQL string arguments.
     - The argument names must match the bind variable names used in the SQL string in content but not in case. For example, if the SQL string is `substr(:INPUT_STRING, 3, 5)`, then you can give the argument name as "input_string".
     - If an argument is used multiple times in an SQL string, then you can declare it once and count it as only one argument.

- `REF` argument

  1. 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.

  2. Use the naming convention for the `REF` arguments.

  - For the reference table name argument, use any dummy name for its column name, for example, "ref-tbl."
  - For the system-generated OID argument, use any dummy name for its column name, such as "sys-OID." Note: Because 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 shortcut later), can be given in any order.

3. Do not set the attribute column names for the object ID to use the shortcut.
   - **Shortcut.** If loading a system-OID-based `REF` column, do not set the column name with a name. The API figures it out. But you must still set other column attributes, such as external data type.
   - If loading a primary-key `REF` column and its primary key consists of multiple columns, the shortcut is not to set their column names. However, you must set other column attributes, such as the external data type.

> **Note:**
>
> If the component column names are NULL, then the API code determines the column names in the position or order in which they were defined for the primary key. So, when you set column attributes other than the name, ensure that the attributes are set for the component columns in the correct order.

## 21.4.3.2 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((void  *)colDesc,
                   OCI_DTYPE_PARAM,
                   (void  *)&flg, (ub4)0,
                   OCI_ATTR_DIRPATH_SID, ctlp->errhp_ctl);
```

## 21.4.3.3 OCI_ATTR_DIRPATH_OID

Indicates that a column is an object table's object ID column.

```
ub1 flg = 1;
sword error;

error = OCIAttrSet((void  *)colDesc,
                   OCI_DTYPE_PARAM,
                   (void  *)&flg, (ub4)0,
                   OCI_ATTR_DIRPATH_OID, ctlp->errhp_ctl);
```

## 21.4.4 Direct Path Function Column Array Handle for Nonscalar Columns

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 nonscalar columns.

Example 21-24 shows how to allocate a child column array for a function context.

**Example 21-24    Allocating 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((void  *)dpfnctx, (void  **)&dpfnca,
                       OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
                       (size_t)0, (void  **)0);
```

This section includes the following topic: OCI_ATTR_NUM_ROWS Attribute.

**See Also:**

Direct Path Function Column Array Handle (OCIDirPathColArray) Attributes

### 21.4.4.1 OCI_ATTR_NUM_ROWS Attribute

This attribute, when used for an `OCI_HTYPE_DIRPATH_FN_COL_ARRAY` (function column array), is retrievable only, and cannot be set by the user.

When the `OCI_ATTR_NUM_ROWS` attribute is called with the function `OCIAttrGet()`, the number of rows allocated for the function column array is returned.

**See Also:**

OCIAttrGet()

## 22

# Object Advanced Topics in OCI

This chapter introduces the OCI facility for working with objects in an Oracle Database.

It also discusses the object navigational function calls, type evolution, and support for XML produced by OCI.

This chapter contains these topics:

- Object Cache and Memory Management
- Object Navigation
- OCI Navigational Functions
- Type Evolution and the Object Cache

## 22.1 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 nonreferenceable objects that cannot be pinned. Only referenceable objects can be pinned.

If an object is being pinned, and an appropriate version exists in the cache, it does not need to be fetched from the server.

Every client program that uses OCI to dereference `REF`s 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 `REF`s 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, and 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.

The object cache is created when the OCI environment is initialized using `OCIEnvCreate()` with `mode` set to `OCI_OBJECT`.

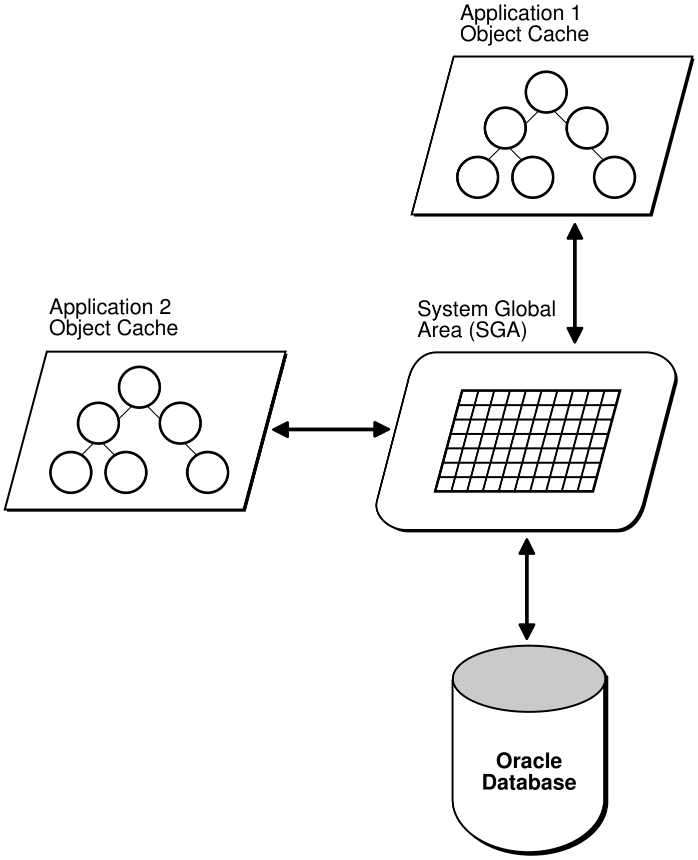
The object cache maintains a fast lookup table for mapping `REF`s to objects. When an application dereferences a `REF` and the corresponding object is not yet cached in the object cache, the object cache automatically sends a request to the server to fetch the object from the database and load it into the object cache.

Subsequent dereferences of the same `REF` are faster because they use 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 an unpin call decrements it. The pin count goes to zero when the object is no longer needed by the application.

The object cache uses a 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 process running against the same server has its own object cache, as shown in Figure 22-1.

**Figure 22-1 Object Cache**


Application 1
Object Cache
Application 2
Object Cache
System Global
Area (SGA)
Oracle
Database

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.

This section includes the following topics:

- Cache Consistency and Coherency
- Object Cache Parameters
- Object Cache Operations
- About Loading and Removing Object Copies
- About Making Changes to Object Copies
- About Synchronizing Object Copies with the Server
- Object Locking
- Commit and Rollback in Object Applications
- Object Duration
- Memory Layout of an Instance

**See Also:**

- About Pinning an Object Copy
- OCIEnvCreate()

## 22.1.1 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 Database 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 occurs 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.

## 22.1.2 Object Cache Parameters

The object cache has two important parameters associated with it, which are attributes of the environment handle.

They include:

- `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 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 that 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 the `OCI_ATTR_CACHE_MAX_SIZE` percentage, using the following algorithm:

```
maximum_cache_size = optimal_size + optimal_size * max_size_percentage / 100
```

Next, represent the algorithm in terms of environment handle attributes.

```
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 8 MB.

The cache size attributes of the environment handle can be set with the `OCIAttrSet()` call and retrieved with the `OCIAttrGet()` function.

**See Also:**

- Environment Handle Attributes for more information
- OCIAttrSet()
- OCIAttrGet()

## 22.1.3 Object Cache Operations

This section describes the most important functions that the object cache provides to operate on object copies.

This section includes the following topics:

- About Pinning and Unpinning
- About Freeing
- About Marking and Unmarking
- About Flushing
- About Refreshing

**See Also:**

OCI Navigational Functions for a list of all the OCI navigational, cache, and object management functions

### 22.1.3.1 About 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.

### 22.1.3.2 About Freeing

What about freeing an object. What does it do.

Freeing an object copy removes it from the cache and frees its memory.

### 22.1.3.3 About 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.

### 22.1.3.4 About 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.

### 22.1.3.5 About Refreshing

Refreshing an object copy in the cache replaces it with the latest value of the corresponding object in the server.

Note that pointers to top-level object memory are valid after a refresh. However, 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 the refresh, but the pointers to second-level memory, such as `scott_p->name`, can be different.

> **See Also:**
>
> OCIObjectRefresh()

## 22.1.4 About Loading and Removing Object Copies

Pin, unpin, and free functions are discussed in this section.

This section includes the following topics:

- About Pinning an Object Copy
- About Unpinning an Object Copy
- About Freeing an Object Copy

### 22.1.4.1 About Pinning an Object Copy

When an application must 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 data type 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 in the cache, if one exists. 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, sales representatives, 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 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 it is important that the program 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 data type 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 the following statement:

```
SELECT VALUE(t) FROM t WHERE REF(t) = :r
```

In this statement, `t` is the object table storing the object, `r` is the `REF`, and the fetched value becomes the value of the object copy in the cache.

Because 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 as *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 have higher concurrency. In a read-committed transaction, when a query is executed multiple times, this type of transaction can produce inconsistent sets of data because it allows changes made by other committed transactions to be seen. This does not happen in serializable transactions.

The object cache model is orthogonal to or independent of the Oracle Database 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()` fails with:

```
ORA-22881: dangling REF
```

**See Also:**

- OCIObjectPin()
- OCIObjectArrayPin()

### 22.1.4.2 About 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 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.

> **See Also:**
>
> - OCIObjectUnpin()
> - OCIObjectPinCountReset()
> - OCICacheUnpin()

## 22.1.4.3 About 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:

- 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 `OCIObjectFree()` to free all object copies in the cache.
- Implicit freeing – if the cache begins 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.

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.

> **See Also:**
>
> - Object Cache Parameters for more information
> - OCIObjectFree()

## 22.1.5 About Making Changes to Object Copies

Functions for marking and unmarking object copies are discussed in this section.

This section includes the following topics:

- About Marking an Object Copy

- About Unmarking an Object Copy

### 22.1.5.1 About 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 is inserted in the server when the object copy is flushed.

If the object copy is updated in the cache, the user must 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 that 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 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 that 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.

**See Also:**

- OCIObjectNew()
- OCIObjectMarkUpdate()
- OCIObjectMarkDelete()

### 22.1.5.2 About Unmarking an Object Copy

A marked object copy can be unmarked in the object cache.

By unmarking a marked object copy, the program ensures that 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 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.

**See Also:**

- OCIObjectUnmark()
- OCICacheUnmark()

## 22.1.6 About Synchronizing Object Copies with the Server

Cache and server synchronization operations (flushing, refreshing) are discussed in this section.

This section includes the following topics:

- About Flushing Changes to the Server
- About Refreshing an Object Copy

### 22.1.6.1 About Flushing Changes to the Server

When the program flushes the object copy, it writes the local changes made to a marked object copy in the cache to the server.

The program can call `OCIObjectFlush()` to flush a single object copy. The program can call `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()`.

After the object copy is flushed, it 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 `OCIObjectFlush()` operation incurs only a single server round-trip even if multiple objects are being flushed.

The callback function (an optional argument to the `OCIObjectFlush()` call) enables an application to flush only dirty objects of a certain type. The application can define a callback that 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 `OCIObjectFlush()`, 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.

However, the `OCI_ATTR_CACHE_ARRAYFLUSH` attribute should be used only if the order in which the objects are flushed is not important. While this attribute is in effect, 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 attribute is enabled, it is not guaranteed that the order in which the objects are marked dirty is preserved.

**See Also:**

- Environment Handle Attributes
- OCIObjectFlush()
- OCICacheFlush()

### 22.1.6.2 About 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 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 22-1 after being refreshed.

**Table 22-1 Object Attributes After a Refresh Operation**

| Object Attribute | Status After Refresh |
|---|---|
| Existent | Set to appropriate value |
| Pinned | Unchanged |
| Flushed | Reset |
| Allocation duration | Unchanged |
| Pin duration | Unchanged |

During the refresh operation, 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 inline attributes of the object. However, the memory for the out-of-line attributes of an object copy can be freed and relocated, because the out-of-line attributes can vary in size.

**See Also:**

- Memory Layout of an Instance for more information about object memory
- OCIObjectRefresh()
- OCICacheRefresh()

## 22.1.7 Object Locking

OCI functions related to object locking are discussed in this section.

This section includes the following topics:

- Lock Options
- About Locking Objects for Update
- About Locking with the NOWAIT Option
- About Implementing Optimistic Locking

### 22.1.7.1 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, which prevents any other user from modifying the object. The lock is released when the transaction commits or rolls back. The different lock options are as follows:

- 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 waits 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 the `OCI_LOCK_X_NOWAIT` option does not wait if the object is currently locked by another user. This is equivalent to executing a `SELECT FOR UPDATE WITH NOWAIT` statement.

### 22.1.7.2 About 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 the following statement:

```
SELECT NULL FROM t WHERE REF(t) = :r FOR UPDATE
```

In this statement, `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.

> **See Also:**
>
> - OCIObjectLock()
> - OCIObjectArrayPin()

## 22.1.7.3 About Locking with the NOWAIT Option

Occasionally, an application attempts to lock an object that is currently locked by another user. In this case, the application is blocked.

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 cannot 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.

> **See Also:**
>
> - OCIObjectLockNoWait()
> - OCIObjectLock()

## 22.1.7.4 About Implementing Optimistic Locking

There are two options available for implementing optimistic locking in an OCI application.

Optimistic locking makes the assumption that a transaction will modify objects in the cache, flush them, and commit the changes successfully.

**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 operation, if a dirty object has been modified by another committed transaction since the beginning of your transaction, a nonserializable transaction error

is returned. If none of the dirty objects has been modified by any other transaction since the beginning of your transaction, then your transaction writes the changes to the database successfully.

> **Note:**
>
> `OCITransCommit()` flushes dirty objects into the database before committing a transaction.

The preceding mechanism effectively implements an optimistic locking model.

**Optimistic Locking Option 2**
Alternately, an application can enable object change detection mode. To do this operation, 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 it attempts 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.

> **See Also:**
>
> OCITransCommit()

## 22.1.8 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.

> **See Also:**
>
> OCITransCommit()

## 22.1.9 Object Duration

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 data type 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; memory is reused when the cache is full.

OCI supports two predefined durations:

- Transaction (`OCI_DURATION_TRANS`)
- Session (`OCI_DURATION_SESSION`)

The *transaction duration* expires when the containing transaction ends (commits or terminates). The *session duration* expires when the containing session or 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.

This section includes the following topic: Durations Example.

**See Also:**

- OCIObjectNew()
- OCIObjectPin()
- OCIObjectUnpin()
- OCICacheUnpin()

### 22.1.9.1 Durations Example

Illustrates the use of the different durations in an application.

Table 22-2 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 is the relative time indicator. The second column indicates the action performed by the database, and the third column indicates the function that 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 22-2 Example of Allocation and Pin Durations**

| Time | Application Action | Function | Object 1 | Object 2 | Object 3 | Object 4 |
|---|---|---|---|---|---|---|
| $T_1$ | Establish connection | - | - | - | - | - |
| $T_2$ | Create object 1 - allocation duration = connection | OCIObjectNew() | Exists | - | - | - |
| $T_5$ | Start Transaction1 | OCITransStart() | Exists | - | - | - |
| $T_6$ | SQL - fetch REF to object 2 | - | Exists | - | - | - |
| $T_7$ | Pin object 2 - pin duration = transaction | OCIObjectPin() | Exists | Pinned | - | - |
| $T_8$ | Process application data | - | Exists | Pinned | - | - |
| $T_9$ | Commit Transaction1 | OCITransCommit() | Exists | Unpinned | - | - |
| $T_{10}$ | Start Transaction2 | OCITransStart() | Exists | - | - | - |
| $T_{11}$ | Create object 3 - allocation duration = transaction | OCIObjectNew() | Exists | - | Exists | - |
| $T_{12}$ | SQL - fetch REF to object 4 | - | Exists | - | Exists | - |
| $T_{13}$ | Pin object 4 - pin duration = connection | OCIObjectPin() | Exists | - | Exists | Pinned |
| $T_{14}$ | Commit Transaction2 | OCITransCommit() | Exists | - | Deleted | Pinned |
| $T_{15}$ | Terminate session1 | OCIDurationEnd() | Exists | - | - | Pinned |
| $T_{16}$ | Start Transaction3 | OCITransStart() | Exists | - | - | Pinned |
| $T_{17}$ | Process application data | - | Exists | - | - | Pinned |
| $T_{18}$ | Commit Transaction3 | OCITransCommit() | Exists | - | - | Pinned |
| $T_{19}$ | Terminate connection | - | Deleted | - | - | Unpinned |

**See Also:**

- The descriptions of OCIObjectNew() and OCIObjectPin() in OCI Navigational and Type Functions for specific information about parameter values that can be passed to these functions
- About Creating Objects for information about freeing up an object's memory before its allocation duration has expired

## 22.1.10 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, some secondary memory chunks.

Consider the `DEPARTMENT` row type defined in Example 22-1.

The C representation of the `DEPARTMENT` is shown in Example 22-2.

Each instance of `DEPARTMENT` has a top-level memory chunk that contains the top-level attributes such as `dep_name`, `budget`, `manager`, and `employees`. The attributes `dep_name` and `employees` are pointers to the additional memory (the secondary memory chunks). The secondary memory is used to contain the 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. In Example 22-2, the top-level memory of the null structure contains the null statuses of the attributes `dep_name`, `budget`, and `manager`, and the atomic nullity of `employees`.

**Example 22-1 Object Type Representation of a Department Row**

```
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 */
```

**Example 22-2 C Representation of a Department Row**

```
struct department
{
OCIString * dep_name;
OCINumber budget;
struct person manager;
OCIArray * employees;
);
typedef struct department department;
```

# 22.2 Object Navigation

This section discusses how OCI applications can navigate through graphs of objects in the object cache.

This section includes the following topic: Simple Object Navigation.

## 22.2.1 Simple Object Navigation

If an application retrieves an object with an attribute that is a `REF` to another object, the application can use OCI calls to traverse the *object graph* and access the referenced instance.

In Example 22-1 and Example 22-2, the object retrieved by the application was a simple object, whose attributes were all scalar values.

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);
```

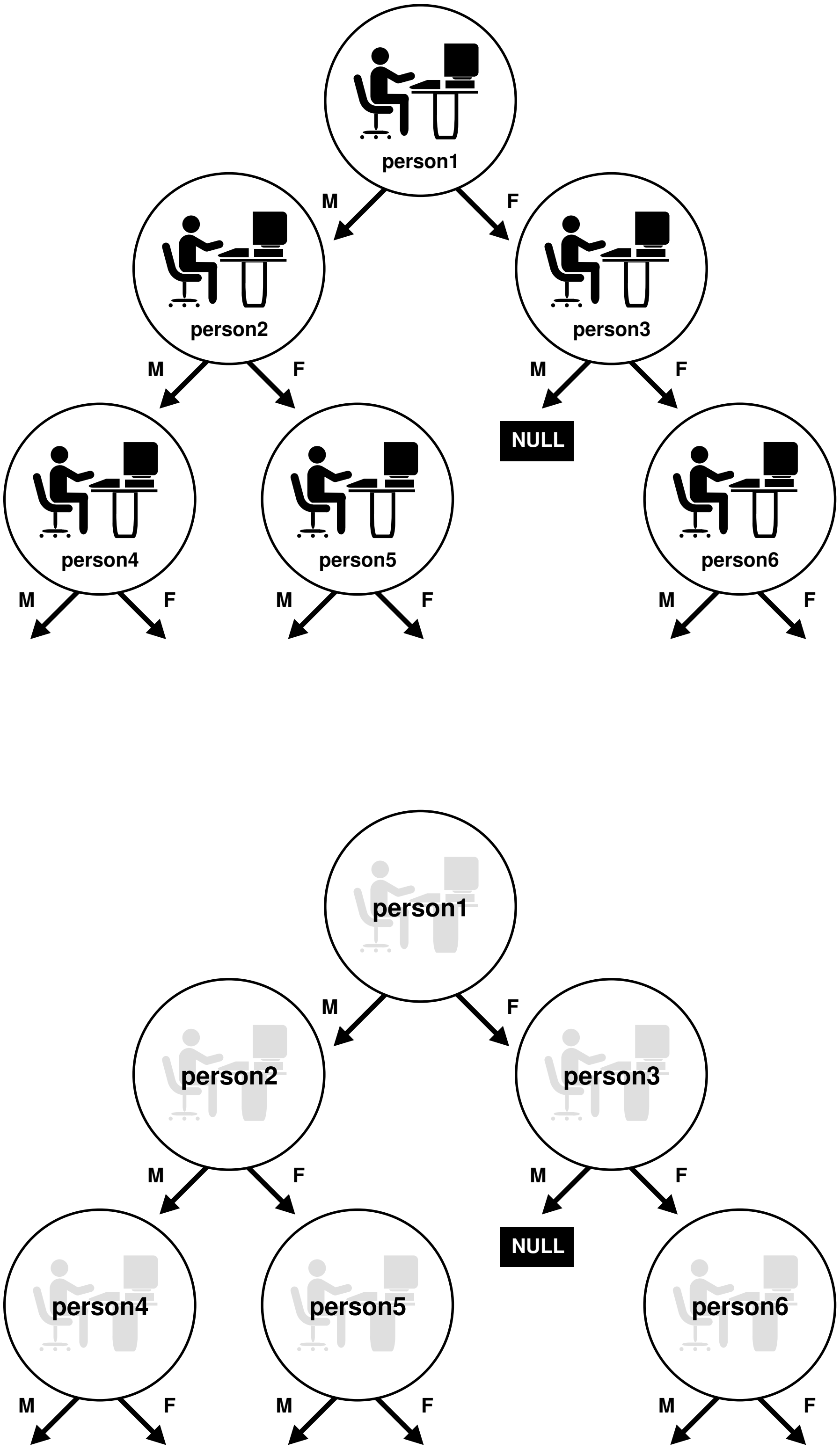
An object table of `person_t` objects is created with the following statement:

```
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 22-2 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. The M and F adjacent to the arrows indicate mother and father, respectively.

**Figure 22-2    Object Graph of person_t Instances**


person1
M
F
person2
person3
M
F
M
F
NULL
person4
person5
person6
M
F
M
F
M
F
person1
M
F
person2
person3
M
F
M
F
NULL
person4
person5
person6
M
F
M
F
M
F

In this case, each object has links to two other instances (M and F) 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 either the previous or next objects or both 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 that 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.

**See Also:**

OCIObjectPin()

# 22.3 OCI Navigational Functions

This section provides a brief summary of the available OCI navigational functions.

The functions are grouped according to their general functionality.

This section includes the following topics:

- About Pin/Unpin/Free Functions
- About Flush and Refresh Functions
- About Mark and Unmark Functions
- About Object Meta-Attribute Accessor Functions
- About Other Functions

Earlier sections of this chapter describe the use of these functions.

The navigational functions follow a naming scheme that uses different prefixes for different types of functionality:

`OCICache*()` – These functions are cache operations.

`OCIObject*()` – These functions are individual object operations.

> **See Also:**
>
> OCI Navigational and Type Functions for more detailed descriptions of each of these functions

## 22.3.1 About Pin/Unpin/Free Functions

Lists and describes the pin, unpin, and free functions.

The functions in Table 22-3 are available to pin, unpin, or free objects.

**Table 22-3 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 |
| OCIObjectPinTable() | Pin a table object with a given duration |
| OCIObjectUnpin() | Unpin an object |

## 22.3.2 About Flush and Refresh Functions

Lists and describes the flush and refresh functions.

The functions in Table 22-4 are available to flush modified objects to the server.

**Table 22-4 Flush and Refresh 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 |

## 22.3.3 About Mark and Unmark Functions

Lists and describes the mark and unmark functions.

The functions in Table 22-5 allow an application to mark or unmark an object by modifying one of its meta-attributes.

**Table 22-5 Mark and Unmark Functions**

| Function | Purpose |
|---|---|
| OCIObjectMarkDeleteByRef() | Mark an object deleted when given a REF |
| OCIObjectMarkUpdate() | Mark an object as updated (dirty) |
| OCIObjectMarkDelete() | Mark an object deleted or delete a value instance |
| OCICacheUnmark() | Unmark all objects in the cache |
| OCIObjectUnmark() | Mark a given object as updated |
| OCIObjectUnmarkByRef() | Mark an object as updated, when given a REF |

## 22.3.4 About Object Meta-Attribute Accessor Functions

Lists and describes the meta-attribute accessor functions.

The functions in Table 22-6 allow an application to access the meta-attributes of an object.

**Table 22-6 Object Meta-Attributes Functions**

| Function | Purpose |
|---|---|
| OCIObjectExists() | Get existence status of an instance |
| OCIObjectGetInd() | Get null structure of an instance |
| OCIObjectIsDirty() | Has an object been marked as updated? |
| OCIObjectIsLocked() | Is an object locked? |

## 22.3.5 About Other Functions

Lists and describes the miscellaneous functions.

The functions in Table 22-7 provide additional object functionality for OCI applications.

**Table 22-7 Other Object Functions**

| Function | Purpose |
|---|---|
| OCIObjectCopy() | Copy one instance to another |
| OCIObjectGetObjectRef() | Return a 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 |

# 22.4 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.

Because 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 the version of the image might differ from the TDO version during the pinning of an object. Then, an error is issued. It is up to you to stop the application or refresh the TDO and repin 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 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, recompile, and relink before executing the program again.

**See Also:**

About Type Evolution

# 23

# OCI Support for XML

Oracle XML DB provides support for storing and manipulating XML instances by using the `XMLType` data type. You can access these XML instances with 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 back end or create new instances on 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 with the C DOM structure `xmldocnode`. You can use this structure for binding, defining, and operating on XML values in OCI statements.

This section includes the following topics:

- XML Context
- XML Data on the Server
- Using OCI XML DB Functions
- OCI Client Access to Binary XML

> **See Also:**
>
> - OCI XML DB Functions for information about the XML support in C
> - *Oracle XML DB Developer’s Guide* for more information about using the C API for XML, including a binary XML example
> - *Oracle XML Developer's Kit Programmer's Guide*for more information about the XML parser for C
> - *Oracle Database XML C API Reference* for information about the DOM C APIs for XML

## 23.1 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);
```

## 23.2 XML Data on the Server

XML data on the server can be operated on with 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.

## 23.3 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.

Example 23-1 is a code fragment of a tested example that shows how to perform operations with the C API.

**Example 23-1    Initializing and Terminating XML Context with a 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 <ocixmldb.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 */
...
}
```

**See Also:**

- OCIXmlDbInitXmlCtx()
- OCIXmlDbFreeXmlCtx()

# 23.4 OCI Client Access to Binary XML

The middle tier and client tiers can produce, consume, and process XML in binary XML format.

The C application fetches data from the XML DB repository, performs updates on the XML using DOM, and stores it back in the database. Or an XML document is created or input on the client and XSLT, XQuery, and other utilities can be used on it. Then the output XML is saved in XML DB.

A client application requires a connection (called a metadata connection) to the metadata repository (typically a back-end database) to fetch token definitions, XML schemas, and DTDs while encoding or decoding a binary XML document.

A repository context is initialized using either a dedicated connection or a connection pool. The connection obtained from the repository context is used to fetch metadata such as token definitions and XML schemas. In contrast, the application also has data connections that are used for the regular transfer of data (including XML data) to and from the database. A repository context is explicitly associated with (one or more) data connections. When XML data is read or written from or to the database using the data connection, the appropriate repository context is accessed during the underlying encode or decode operations. As required, the metadata connection is used to fetch the metadata from the repository.

This section includes the following topics:

- Accessing XML Data from an OCI Application
- Repository Context
- Create Repository Context from a Dedicated OCI Connection
- Create Repository Context from a Connection Pool
- About Associating Repository Context with a Data Connection
- About Setting XMLType Encoding Format Preference
- Example of Using a Connection Pool

## 23.4.1 Accessing XML Data from an OCI Application

Your C application can use OCI to access persistent XML in the database and the Unified XML C API to operate on the fetched XML data.

The following steps are taken by a client application:

1. Create the usual OCI handles such as `OCIEnv`, `OCISvcCtx`, and `OCIError`.
2. Create one or more repository contexts to fetch the binary XML metadata.
3. Associate the repository context with the data connection.
4. Bind or define (`xmldocnode`) variables into the select, insert, and update statements.
5. Execute the select, insert, or update statement to fetch or store the XML document. At this point, the client OCI libraries interact with the database back end to fetch the needed XML Schemas, DTDs, token definitions, and so on.
6. Use the Unified C API to operate on the XML data (DOM).

## 23.4.2 Repository Context

`OCIBinXmlReposCtx` is the repository context data structure.

The client application creates this context by providing the connection information to the metadata repository. An application can create multiple repository contexts to connect to multiple token repositories. A repository context is explicitly associated with a data connection (`OCISvcCtx`). When the system must fetch metadata to encode or decode data to or from a data connection, it accesses the appropriate metadata.

It is recommended that applications create one repository context per `OCIEnv`. This allows better concurrency for multithreaded applications.

The repository context can be created out of a dedicated OCI connection or an OCI connection pool.

## 23.4.3 Create Repository Context from a Dedicated OCI Connection

`OCIBinXmlCreateReposCtxFromConn()` creates a repository context using the specified dedicated OCI connection.

The OCI connection is only to be used for metadata access and should not be used in any other scenarios by the application. Also note that the access to this connection is serialized; that is, if multiple threads try to use the same connection, access is limited to one thread at a time. For scalability reasons, it is recommended that applications create a repository context using a connection pool, as described in the next section.

Note: You can also potentially pass in the same connection as the one being used for data. However, this might result in an error in certain cases where the client system attempts to contact the metadata repository while part of another operation (such as select or insert).

**See Also:**

OCIBinXmlCreateReposCtxFromConn()

## 23.4.4 Create Repository Context from a Connection Pool

`OCIBinXmlCreateReposCtxFromCPool()` creates a repository context from a connection pool.

When the application accesses the back-end repository, any available connection from the pool is used. Further, this connection is released back to the pool as soon as the metadata operation is complete. Connection pools are highly recommended for multithreaded application scenarios. Different threads can use different connections in the pool and release them as soon as they are done. This approach allows for higher scalability and concurrency with a smaller number of physical connections.

**See Also:**

OCIBinXmlCreateReposCtxFromCPool()

## 23.4.5 About Associating Repository Context with a Data Connection

`OCIBinXmlSetReposCtxForConn()` associates a repository context with a data connection described by `OCISvcCtx *`.

Multiple data connections can share the same repository context, but access to the repository can be serialized (if it is based on a dedicated connection). When the system must fetch the metadata for encode or decode operations, it looks up the appropriate repository connection from the `OCIEnv`, `OCISvcCtx` pair and uses it to fetch the metadata required.

**See Also:**

OCIBinXmlSetReposCtxForConn()

## 23.4.6 About Setting XMLType Encoding Format Preference

By default, XML data sent to the database is encoded in one of these possible formats (text, object-relational, or binary XML) based on certain internal criteria such as the source format (if it was read from the DB). `OCIBinXmlSetFormatPref()` provides an explicit mechanism to set the preference for encoding format. In the future, the default format can be binary XML, but this function could be used to override it if needed.

> **See Also:**
>
> OCIBinXmlSetFormatPref()

## 23.4.7 Example of Using a Connection Pool

Creating a repository context from a connection pool and associating the repository context with a data connection is shown in this example in the XML DB documentation.

The database is local and the test is in single-threaded mode.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information about using OCI and the C API for XML with Oracle XML DB

# 24

# 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:

- What Is the Object Type Translator?
- OTT Command Line
- Intype File
- OTT Data Type Mappings
- Outtype File
- About Using OTT with OCI Applications
- OTT Reference

## 24.1 What Is the Object Type Translator?

The Object Type Translator (OTT) assists in the development of C language applications that make use of user-defined types in an Oracle database.

With 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 must be able to represent the data in a host language format. This is accomplished by representing object types as C structs. Although it is possible for a programmer to code struct declarations by hand to represent database object types, this can be very time-consuming and error-prone if many types are involved. OTT obviates the need for such manual coding by automatically generating appropriate struct declarations. In OCI, the application also must call an initialization function generated by OTT.

In addition to creating structs that represent stored data types, OTT generates parallel indicator structs that indicate whether an object type or its fields are `NULL`.

The Object Type Translator (OTT) converts database definitions of object types and named collection types into C struct declarations that 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 that invokes OTT:

```
ott userid=scott 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`. The user is prompted for the password.

The implementation file (`demov.c`) contains the function to initialize the type version table with information about the user-defined types translated.

Later sections of this chapter describe each of these parameters in more detail.

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 OTT.

Once 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. Suppose for example, that the employee type listed in the intype file was defined as shown in Example 24-1.

Then the header file generated by OTT (`demo.h`) includes, among other items, the declarations shown in Example 24-2.

Example 24-3 shows what a sample implementation file (`demov.c`) produced by this command contains.

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 lowercase because of the line `CASE=lower` in the intype file.

The data types that appear in the struct declarations (for example, `OCIString`, `OCIInd`) are special data types.

The remaining sections of this chapter discuss the use of OTT with OCI, followed by a reference section that describes command-line syntax, parameters, intype file structure, nested `#include` file generation, schema names usage, default name mapping, and restrictions.

**Example 24-1 Definition of the Employee Object Type Listed in the Intype File**

```
CREATE TYPE emptype AS OBJECT
(
    name       VARCHAR2(30),
    empno      NUMBER,
    deptno     NUMBER,
    hiredate   DATE,
    salary     NUMBER
);
```

**Example 24-2 Contents of the Generated Header File demo.h**

```
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;
```

**Example 24-3 Contents of the demov.c File**

```
#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;
}
```

This section includes the following topics:

- About Creating Types in the Database
- About Invoking OTT

**See Also:**

OTT Data Type Mappings for more information about these types

## 24.1.1 About 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:**

Oracle Database SQL Language Reference for information about the `CREATE TYPE` statement

## 24.1.2 About 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 takes 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`.) However, 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.

This section includes the following topics:

- Command Line
- Configuration File
- INTYPE File

### 24.1.2.1 Command Line

Parameters (also called options) set on the command line override any set elsewhere.

**See Also:**

OTT Command Line

### 24.1.2.2 Configuration File

A configuration file is a text file that contains OTT parameters.

Each nonblank 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 is used. Whitespace is not allowed on any nonblank line of a configuration file.

A configuration file can 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.

### 24.1.2.3 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:**
>
> Intype File

## 24.2 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.

This section includes the following topic: OTT Command-Line Invocation Example.

> **See Also:**
>
> About Using the Object Type Translator for Windows

### 24.2.1 OTT Command-Line Invocation Example

Shows how to invoke OTT from the command line.

Example 24-4 shows how to invoke OTT from the command line.

> **Note:**
>
> No spaces are permitted around the equal sign (=).

**Example 24-4    Invoking OTT from the Command Line**

```
ott userid=bren intype=demoin.typ outtype=demoout.typ code=c \
    hfile=demo.h initfile=demov.c
```

The following sections describe the elements of the command line used in this example.

This section includes the following topics:

- OTT
- USERID
- INTYPE
- OUTTYPE
- CODE
- HFILE
- INITFILE

> **See Also:**
>
> OTT Reference for a detailed discussion of the various OTT command-line options

## 24.2.1.1 OTT

Causes OTT to be invoked.

It must be the first item on the command line.

## 24.2.1.2 USERID

Specifies the database connection information that OTT uses.

In Example 24-4, OTT attempts to connect with user name `bren` and is then prompted for the password.

## 24.2.1.3 INTYPE

Specifies the name of the intype file that is used.

In Example 24-4, the name of the intype file is specified as `demoin.typ`.

## 24.2.1.4 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 that is translated, including its version string, and the header file to which its C representation was written.

In Example 24-4, the name of the outtype file is specified as `demoout.typ`.

**Note:**

If the file specified by the outtype keyword 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.

## 24.2.1.5 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 the initialization function defined in the `INITFILE` file depends on whether `KR_C` is used. If the `INITFILE` option is not used, all three options are equivalent.

## 24.2.1.6 HFILE

Specifies the name of the C header file to which the generated structs should be written.

In Example 24-4, the generated structs are stored in a file called `demo.h`.

**Note:**

If the file specified by the `hfile` keyword exists, it is 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 does not actually write to the file. This preserves the modification time of the file so that Linux and UNIX `make` and similar facilities on other operating systems do not perform unnecessary recompilations.

## 24.2.1.7 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 exists, it is 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 does not actually write to the file. This preserves the modification time of the file so that Linux and UNIX `make` and similar facilities on other operating systems do not perform unnecessary recompilations.

## 24.3 Intype File

When OTT runs, the intype file tells OTT which database types should be translated.

It can also control the naming of the generated structs. The intype file can be a user-created file, or it can 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.

Example 24-5 shows a simple user-created intype file.

Example 24-5 is further described as follows.

The first line, with the `CASE` keyword, indicates that generated C identifiers should be in lowercase. 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 lowercase.

In the lines that 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 is used as the C identifier name, except that the `CASE` option is observed, and any character that cannot be mapped to a legal C identifier character is replaced by an underscore. Reasons for translating a type or attribute name include the following:

- 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 can happen, for example, if the program uses two types with the same name from different schemas.
- The programmer prefers a different name.

OTT may need to translate additional types that are not listed in the intype file. This is because 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 does not generate types that are not specified in the intype file.

A normal case-insensitive SQL identifier can be spelled in any combination of uppercase and lowercase in the intype file, and is not enclosed within quotation marks.

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 enclosed within quotation marks 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"`). Therefore, when a name is enclosed within quotation marks, the name enclosed within quotation marks must be in uppercase 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 enclosed within quotation marks, OTT reports a syntax error in the intype file.

**Example 24-5    Contents 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
```

**See Also:**

- CASE
- Structure of the Intype File for a more detailed specification of the structure of the intype file and the available options

## 24.4 OTT Data Type 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 data types of the attributes are mapped to types that are used in Oracle's object data types. The data types found in Oracle Database include a set of predefined, primitive types. These data types provide for the creation of user-defined types, such as object types and collections.

Oracle Database also includes a set of predefined types that are used to represent object type attributes in C structs. As an example, consider the object type definition in Example 24-6, and its corresponding OTT-generated struct declarations in Example 24-7.

The OTT output, assuming `CASE=LOWER` and no explicit mappings of type or attribute names, is shown in Example 24-7.

The data types in the struct declarations—`OCIString`, `OCINumber`, `OCIDate`, and `OCIInd`—are used here to map the data types of the object type attributes. The `NUMBER` data type of the `empno` attribute maps to the `OCINumber` data type, for example. These data types can also be used as the types of bind and define variables.

**Example 24-6    Object Type Definition for Employee**

```
CREATE TYPE employee AS OBJECT
(   name        VARCHAR2(30),
    empno       NUMBER,
    deptno      NUMBER,
    hiredate    DATE,
    salary$     NUMBER);
```

**Example 24-7    OTT-Generated Struct Declarations**

```
struct employee
{   OCIString * name;
    OCINumber empno;
    OCINumber deptno;
    OCIDate   hiredate;
    OCINumber salary_;
};
typedef struct emp_type emp_type;
struct employee_ind
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd empno;
    OCIInd deptno;
    OCIInd hiredate;
    OCIInd salary_;
}
typedef struct employee_ind employee_ind;
```

This section includes the following topics:

- About Mapping Object Data Types to C
- OTT Type Mapping Example
- Null Indicator Structs
- OTT Support for Type Inheritance

**See Also:**

Null Indicator Structs for an explanation of the indicator struct (`struct employee_ind`)

## 24.4.1 About Mapping Object Data Types to C

This section describes the mappings of Oracle object attribute types to C types generated by OTT.

The previous section OTT Type Mapping Example includes examples of many of these different mappings. Table 24-1 lists the mappings from types that you can use as attributes to object data types that are generated by OTT and the corresponding OCI type code values.

**Table 24-1 Object Data Type Mappings for Object Type Attributes**

| Object Attribute Types | C Mapping | OCITypeCode Values |
|---|---|---|
| BFILE | OCIBFileLocator* | OCI_TYPECODE_BFILE |
| BLOB | OCILobLocator * or OCIBlobLocator * | OCI_TYPECODE_BLOB |
| CHAR(N), CHARACTER(N), NCHAR(N) | OCIString * | OCI_TYPECODE_CHAR (n),<br>OCI_TYPECODE_NCHAR |
| CLOB, NCLOB | OCILobLocator * or OCIClobLocator * | OCI_TYPECODE_CLOB,<br>OCI_TYPECODE_NCLOB |
| DATE | OCIDate | OCI_TYPECODE_DATE |
| ANSI DATE | OCIDateTime * | OCI_TYPECODE_TIMESTAMP |
| TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE | OCIDateTime * | OCI_TYPECODE_TIMESTAMP,<br>OCI_TYPECODE_TIMESTAMP_TZ,<br>OCI_TYPECODE_TIMESTAMP_LTZ |
| INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND | OCIInterval * | OCI_TYPECODE_INTERVAL_YM,<br>OCI_TYPECODE_INTERVAL_DS |
| DEC, DEC(N), DEC(N,N) | OCINumber | OCI_TYPECODE_DECIMAL (p) |
| DECIMAL, DECIMAL(N), DECIMAL(N,N) | OCINumber | OCI_TYPECODE_DECIMAL (p) |
| FLOAT, FLOAT(N), DOUBLE PRECISION | OCINumber | OCI_TYPECODE_FLOAT (b) |
| BINARY_FLOAT | float | OCI_TYPECODE_BFLOAT |
| BINARY_DOUBLE | double | OCI_TYPECODE_BDOUBLE |
| INT, INTEGER, SMALLINT | OCINumber | OCI_TYPECODE_INTEGER,<br>OCI_TYPECODE_SMALLINT |
| Nested Object Type | C name of the nested object type | OCI_TYPECODE_OBJECT |
| Nested Table | OCITable * | OCI_TYPECODE_TABLE |
| NUMBER, NUMBER(N), NUMBER(N,N) | OCINumber | OCI_TYPECODE_NUMBER (p, s) |
| NUMERIC, NUMERIC(N), NUMERIC(N,N) | OCINumber | OCI_TYPECODE_NUMBER (p, s) |
| RAW(N) | OCIRaw * | OCI_TYPECODE_RAW |
| REAL | OCINumber | OCI_TYPECODE_REAL |
| REF | OCIRef * | OCI_TYPECODE_REF |
| VARCHAR(N) | OCIString * | OCI_TYPECODE_VARCHAR (n) |
| VARCHAR2(N), NVARCHAR2(N) | OCIString * | OCI_TYPECODE_VARCHAR2 (n),<br>OCI_TYPECODE_NVARCHAR2 |
| VARRAY | OCIArray * | OCI_TYPECODE_VARRAY |
| collections | OCIColl * | OCI_TYPECODE_NAMEDCOLLECTION |

> **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 an example, see 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 data types to which OTT maps non-object database attribute types are structures, which, except for `OCIDate`, are opaque.

## 24.4.2 OTT Type Mapping Example

Shows the various type mappings created by OTT when given the database types.

Example 24-9 demonstrates the various type mappings created by OTT when given the database types shown in Example 24-8.

The intype file includes the following:

```
CASE = LOWER
TYPE many_types
```

OTT generates the C structs shown in Example 24-9.

> **Note:**
>
> Comments are provided in Example 24-9 to help explain the structs. These comments are not part of actual OTT output.

Notice that although 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` has the default value of `TRUE`. As described in that section, when `TRANSITIVE=TRUE`, OTT automatically translates any types that are used as attributes of a type being translated, to complete the translation of the listed type.

This is not the case for types that 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;
   ...
}
```

### Example 24-8    Object Type Definitions for the OTT Type Mapping Example

```
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));
```

### Example 24-9    Various Type Mappings Created by OTT from Object Type Definitions

```
#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
```

**See Also:**

TRANSITIVE

## 24.4.3 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 Example 24-9.

```
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`).

The `varrays` and `nested tables` contain the `NULL` information for their elements.

The data type for all other elements of a `NULL` indicator struct is `OCIInd`.

**See Also:**

NULL Indicator Structure for more information about atomic nullity

## 24.4.4 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, suppose that you have a type `Person_t`, with subtype `Student_t` and subtype `Employee_t`, as shown in Example 24-10.

Suppose that you also have an intype file with the content shown in Example 24-11.

Then, OTT generates the C structs for `Person_t`, `Student_t`, and `Employee_t`, and their `NULL` indicator structs, as shown in Example 24-12.

The preceding C mapping convention allows simple upcasting 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;   /* upcasting */
```

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).

### Example 24-10 Object Type and Subtype Definitions

```
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));
```

### Example 24-11 Contents of the Intype File

```
CASE=SAME
TYPE EMPLOYEE_T
TYPE STUDENT_T
TYPE PERSON_T
```

### Example 24-12 OTT Generates C Structs for the Types and 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
```

This section includes the following topic: Substitutable Object Attributes.

### 24.4.4.1 Substitutable Object Attributes

For attributes of `NOT FINAL` types (potentially substitutable), the embedded attribute is represented as a pointer.

Consider a type `Book_t` created as follows:

```
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 as follows:

```
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 must change. The new mapping is generated by running OTT again.

**See Also:**

OCIObjectGetInd()

## 24.5 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 that 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, suppose that you have a simple intype file, as shown in Example 24-13, which was used in Example 24-5.

The user has chosen to specify the case for the OTT-generated C identifiers, and has provided a list of types to be translated. In two of these types, naming conventions are specified.

Example 24-14 shows what the outtype file might look like after running OTT.

When examining the contents of the outtype file, you might discover types listed that were not included in the intype specification. For example, suppose that the intype file only specified that the `person` type was to be translated as follows:

```
CASE = LOWER
TYPE PERSON
```

However, because the definition of the `person` type includes an attribute of type `address`, the outtype file includes 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.

### Example 24-13 Contents of an 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
```

### Example 24-14 Contents of the Outtype File 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
```

## 24.6 About Using OTT with OCI Applications

An OCI application that accesses objects in an Oracle server can use C header and implementation files that have been generated by OTT.

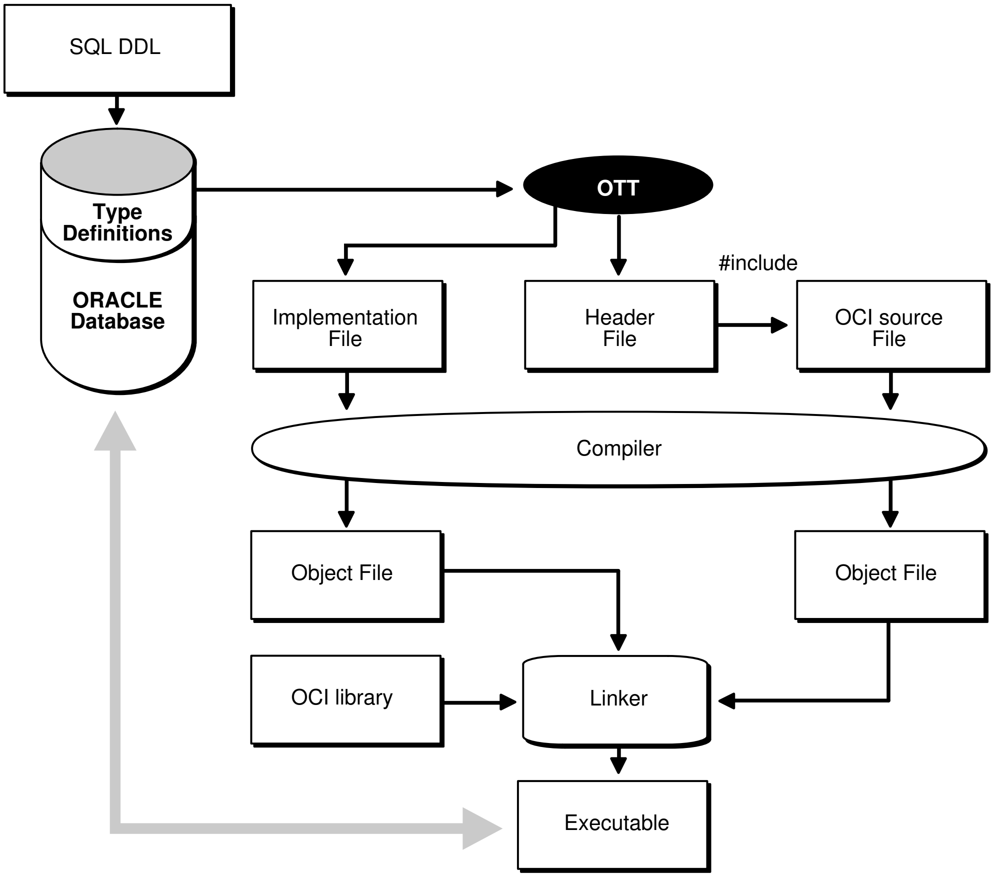
The header file is incorporated into the OCI code with an `#include` statement.

Once the header file has been included, the OCI application can access and manipulate object data in the host language format.

Figure 24-1 shows the steps involved in using OTT with 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 database.

**Figure 24-1 Using OTT with OCI**


SQL DDL
Type Definitions
ORACLE Database
OTT
#include
Implementation File
Header File
OCI source File
Compiler
Object File
Object File
OCI library
Linker
Executable

This section includes the following topics:

- About Accessing and Manipulating Objects with OCI
- Calling the Initialization Function
- Tasks of the Initialization Function

## 24.6.1 About 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 that appear in the 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 data type mapping and manipulation functions that 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 data type.
- `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.

> **See Also:**
>
> - OCIStringSize()
> - OCINumberAdd()
> - OCILobIsEqual()
> - OCIRawPtr()
> - OCICollAppend()
> - OCITableFirst()
> - OCIRefIsNull()

## 24.6.2 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 can specify a name for the initialization function when you invoke OTT with the `INITFUNC` option, or you can 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 data types 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.

Suppose that you have an intype file, `ex2c.typ`, containing the content shown in Example 24-15.

Then you invoke OTT from the command line and specify the initialization function, as shown in Example 24-16.

OTT generates the `ex2cv.c` file with the contents shown in Example 24-17.

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.

**Example 24-15 Content of an Intype File Named ex2c.typ**

```
TYPE BREN.PERSON
TYPE BREN.ADDRESS
```

**Example 24-16 Invoking OTT and Specifying the Initialization Function**

```
ott userid=bren intype=ex2c outtype=ex2co hfile=ex2ch.h initfile=ex2cv.c
```

**Example 24-17 Content of an OTT-Generated File Named 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;
}
```

**See Also:**

OCIEnvCreate()

### 24.6.3 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 data type.

The type-version table is used by the Oracle database type manager to determine which version of a type a particular program uses. Different initialization functions generated by OTT at different times can add some of the same types to the type version table. When a type is added more than once, Oracle Database ensures that 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 Database, each type has only one version. Initialization of the type version table is required only for compatibility with future releases of Oracle Database.

## 24.7 OTT Reference

Parameters that can appear on the OTT command line or in a `CONFIG` file control the behavior of OTT. Certain parameters can 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
- OTT Command on Microsoft Windows

The following conventions are used in this section 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 lowercase monospaced font in examples and headings, but are printed in uppercase 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.

## 24.7.1 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 you develop OCI applications that use objects.

An OTT command-line statement consists of the keyword `OTT`, followed by a list of OTT parameters.

The parameters that can appear on an OTT command-line statement are as follows:

```
[userid=username/password[@db_name]]

[intype=filename]

outtype=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. Only the `OUTTYPE` and `CODE` parameters are always required.

The `HFILE` parameter is almost always used. If omitted from the command line, `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 is 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 is translated. The OTT parameter descriptions are described in the sections that follow.

The following is an example of an OTT command-line statement (you are prompted for the password):

```
ott userid=marc intype=in.typ outtype=out.typ code=c hfile=demo.h\
   errtype=demo.tls case=lower
```

The following sections describe each of the OTT command-line parameters.

**See Also:**

OTT Parameters

## 24.7.2 OTT Parameters

Enter parameters on the OTT command line using the following format:

```
parameter=value
```

In this format, `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 by 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.

This section includes the following OTT parameters:

- USERID
- INTYPE
- OUTTYPE
- CODE
- INITFILE
- INITFUNC
- HFILE
- CONFIG
- ERRTYPE
- CASE
- SCHEMA_NAMES
- TRANSITIVE
- URL

### 24.7.2.1 USERID

The `USERID` parameter specifies the database 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]
```

The `USERID` parameter is optional. If it is omitted, OTT automatically attempts to connect to the default database as user `OPS$username`, where `username` is the user's operating system user name. If this is the first parameter, "`USERID=`" and the password and the database name can be omitted, as shown here:

```
ott username ...
```

For security purposes, when you enter only the user name you are prompted for the rest of the entry.

The `username` and `db_name` (schema name) combination is appended and used as a key in the `OCIContextGetValue()` call, which supports a maximum length of 64 bytes in Oracle Database 12*c* Release 1 (12.1). In Oracle Database 12*c* Release 2 (12.2) this max length is extended to support 243 bytes, not including 1 character for the period (.).

### 24.7.2.2 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=`" can be omitted if `USERID` and `INTYPE` are the first two parameters, in that order, and "`USERID=`" is omitted. If the `INTYPE` parameter is not specified, all types in the user's schema are translated.

```
ott username 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.

If the file name on the command line or in the intype file does not include an extension, an operating system-specific extension such as "`TYP`" or "`.typ`" is added.

**See Also:**

Structure of the Intype File for a description of the format of the intype file

## 24.7.2.3 OUTTYPE

The `OUTTYPE` parameter specifies the name of a file into which OTT writes type information for all the object data types it processes.

This includes all types explicitly named in the intype file, and can include additional types that are translated because they are used in the declarations of other types that must be translated (if `TRANSITIVE=TRUE`). This file must 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` parameter 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.

The parameter `OUTTYPE` must be specified.

If the file name on the command line or in the outtype file does not include an extension, an operating system-specific extension such as "`TYP`" or "`.typ`" is added.

## 24.7.2.4 CODE

This is the desired host language for OTT output, which is 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.

## 24.7.2.5 INITFILE

The `INITFILE` parameter specifies the name of the file where the OTT-generated initialization file is to be written.

The initialization function is not generated if this parameter is omitted.

For Pro*C/C++ programs, the `INITFILE` is not necessary, because the SQLLIB runtime library performs the necessary initializations. An OCI program user must compile and link the `INITFILE` files, and must call the initialization file functions 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, an operating system-specific extension such as "`C`" or "`.c`" is added.

```
initfile=filename
```

### 24.7.2.6 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
```

### 24.7.2.7 HFILE

The `HFILE` parameter specifies 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, an operating system-specific extension such as "`H`" or "`.h`" is added.

```
hfile=filename
```

### 24.7.2.8 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 an 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.

### 24.7.2.9 ERRTYPE

If the `ERRTYPE` parameter is supplied, OTT writes a listing of the intype file to the `ERRTYPE` file, along with all informational and error messages.

Informational and error messages are sent to the standard output whether `ERRTYPE` parameter is specified or not.

Essentially, the `ERRTYPE` file is a copy of the intype file with error messages added. In most cases, an error message includes a pointer to the text that caused the error.

If the file name of an `ERRTYPE` on the command line or in the intype file does not include an extension, an operating system-specific extension such as "`TLS`" or "`.tls`" is added.

```
errtype=filename
```

## 24.7.2.10 CASE

This `CASE` 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 lowercase, 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 occurs after a legal identifier has been generated.

Note that the case of the C struct identifier for a type specifically mentioned in the `INTYPE` parameter option is the same as its case in the intype file. For example, if the intype file includes the following line:

```
TYPE Worker
```

Then OTT generates the following line:

```
struct Worker {...};
```

However, suppose that the intype file is written as follows:

```
TYPE wOrKeR
```

Then OTT generates the following line, following the case specified in the intype file.

```
struct wOrKeR {...};
```

Case-insensitive SQL identifiers not mentioned in the intype file appear in uppercase if `CASE=SAME`, and in lowercase if `CASE=OPPOSITE`. A SQL identifier is case-insensitive if it was not enclosed in quotation marks when it was declared.

## 24.7.2.11 SCHEMA_NAMES

The `SCHEMA_NAMES` parameter 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:**

SCHEMA_NAMES Usage

### 24.7.2.12 TRANSITIVE

The `TRANSITIVE` parameter takes the values `TRUE` (the default) or `FALSE`.

It 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.

### 24.7.2.13 URL

For the `URL` parameter, 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 Database installation.

To specify the JDBC Thin driver (the Java driver for client-side use without an Oracle Database installation), use the following `URL` parameter syntax:

```
URL=jdbc:oracle:thin:@host:port:sid
```

The `host` is the name of the host on which the database is running, `port` is the port number, and `sid` is the Oracle SID.

## 24.7.3 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 the following, then additional parameters are read from the configuration file `filename`:

```
config=filename
```

In addition, parameters are also read from a default configuration file in an 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 online 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` can also appear in the intype file. The 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.

## 24.7.4 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 can 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.

This section includes the following topic: Intype File Type Specifications.

**See Also:**

Outtype File for an example of a simple user-defined intype file, and of the full outtype file that OTT generates from it

### 24.7.4.1 Intype File Type Specifications

A type specification in the intype file names an object data type that is to be translated.

A type specification in the outtype file names an object data type 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
```

The `schema_name` is the name of the schema that owns the given object data type, 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 as follows:

- `type_name` is the name of an Oracle Database object data type.
- `type_identifier` is the C identifier used to represent the type. If `type_identifier` is omitted, the default name mapping algorithm is used.
- `version_string` is the version string of the type that 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 can be used as the intype file when OTT is executed later. The version string does not affect the operation of OTT, but is eventually used to select the version of the object data type that should be used in the running program.
- `hfile_name` is the name of the header file in which the declarations of the corresponding struct or class appear. If `hfile_name` is omitted, the file named by the command-line `HFILE` parameter is used if a declaration is generated.
- `member_name` is the name of an attribute (data member) that is to be translated to the `identifier`.
- `identifier` is the C identifier used to represent the attribute in the user program. Identifiers can be specified in this way for any number of attributes. The default name mapping algorithm is used for the attributes that are not mentioned.

An object data type 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, OTT writes the translation of the required type to the same file or files 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, OTT writes the translation of the required type to the global `HFILE` file.

**See Also:**

Default Name Mapping for more information about naming the intype file related to the `version_string` component.

## 24.7.5 Nested Included File Generation

Every `HFILE` generated by OTT uses `#include` directives to include other necessary files and `#define` directives to define a symbol constructed from the name of the file, which can be used to determine if the `HFILE` has been included.

Consider, for example, a database with the types shown in Example 24-18.

The intype file content is shown in Example 24-19.

If you invoke OTT with the command shown in Example 24-20, then it generates the header files shown in Example 24-21 and Example 24-22.

The content of the header file `tott95b.h` is shown in Example 24-21.

The content of the header file tott95a.h is shown in Example 24-22.

In Example 24-21, the symbol `TOTT95B_ORACLE` is defined first so that the programmer can conditionally include `tott95b.h` without having to worry whether `tott95b.h` depends on the include file using the construct, as shown in Example 24-23.

Using this technique, the programmer can 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 can use. This is the only case in which OTT uses angle brackets in an `#include` directive.

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 generates the necessary `#includes` directives.

Note that OTT uses quotation marks in this `#include` directive. When a program including `tott95b.h` is compiled, the search for `tott95a.h` begins where the source program was found, and thereafter follows an implementation-defined search rule. If `tott95a.h` cannot be found in this way, a complete file name (for example, a Linux or UNIX absolute path name beginning with /) should be used in the intype file to specify the location of `tott95a.h`.

### Example 24-18 Object Type Definition to Demonstrate How OTT Generates Include Files

```
create type px1 AS OBJECT (col1 number, col2 integer);
create type px2 AS OBJECT (col1 px1);
create type px3 AS OBJECT (col1 px1);
```

### Example 24-19 Content of the Intype File

```
CASE=lower
type pxl
  hfile tott95a.h
type px3
  hfile tott95b.h
```

### Example 24-20 Invoking OTT from the Command Line

```
ott scott tott95i.typ outtype=tott95o.typ code=c
```

### Example 24-21 Content of the Header File tott95b.h

```
#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
```

**Example 24-22    Content of the Header File tott95a.h**

```
#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
```

**Example 24-23    Construct to Use to Conditionally Include the Header File tott95b.h**

```
#ifndef TOTT95B_ORACLE
#include "tott95b.h"
#endif
```

## 24.7.6 SCHEMA_NAMES Usage

The `SCHEMA_NAMES` 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.

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 than 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 must be generated because of type dependencies is 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 is always 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 runtime to ensure 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 is found in the named schema during program execution. If the type appears without a schema name, the type is found in the default schema to which the program connects, which can be different from the default schema that OTT used.

This section includes the following topic: Example: Schema_Names Usage.

## 24.7.6.1 Example: Schema_Names Usage

Shows examples of using `SCHEMA_NAMES` parameter.

Suppose that `SCHEMA_NAMES` is set to `FROM_INTYPE`, and the intype file reads as follows:

```
TYPE Person
TYPE david.Dept
TYPE sam.Company
```

Then the Pro*C/C++ application that uses the OTT-generated structs uses 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 uses the same type (`david.Person`) that OTT used. If OTT connected to schema `david` but the application connects to schema `jana`, the application uses 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`.

In contrast, the application uses type `david.Dept` regardless of to which schema the application is connected. If this is the behavior that 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`.

However, the intype file does 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, then "`TYPE david.Person`" and "`TYPE david.Address`" appear in the outtype file. If "`Type Person`" appeared in the intype file, then "`TYPE Person`" and "`TYPE Address`" 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 specify the following in the intype file:

```
TYPE      david.Address
```

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.

## 24.7.7 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 can 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 by using the following steps:

1. 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. 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. 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 can, for example, translate accented single-byte characters such as "o" with an umlaut or "a" with an accent grave to "o" or "a", and can translate a multibyte letter to its single-byte equivalent. Name translation typically fails 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 does not detect a naming clash caused by two or more database identifiers being mapped to the same C name, nor does it detect a naming problem where a database identifier is mapped to a C keyword.

## 24.7.8 OTT Restriction on File Name Comparison

Currently, 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 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` directive if the two files are the same, and two `#includes` directives if the files are different. In practice, though, it would conclude that the two files are different, and would generate two `#includes` directives, 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 is included. If `foo1.h` and `/private/elias/foo1.h` are the same file, a redundant `#include` directive is 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.

## 24.7.9 OTT Command on Microsoft Windows

OTT executable on Microsoft Windows in the current release is `ott.bat`, instead of `ott.exe` as in the earlier releases.

This may break Windows batch scripts, as the scripts exit immediately after executing ott. To fix this problem, OTT should be invoked as follows, in Windows batch scripts:

```
call ott [arguments]
```

> **Note:**
>
> `ORACLE_HOME\precomp\admin\ott.exe` can be used until the scripts are fixed, as an intermediate solution. However, this intermediate solution will not be provided in future releases.

# 25

# Oracle Database Access C API

This chapter begins to describe the Oracle Database Access C API and in particular the OCI relational functions for C.

This chapter 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 Database installation. For additional information, see OCI Demonstration Programs.

This chapter contains these topics:

- Introduction to the Relational Functions
- Connect, Authorize, and Initialize Functions
- Handle and Descriptor Functions
- Bind, Define, and Describe Functions

## 25.1 Introduction to the Relational Functions

Introduces OCI relational function calls.

This chapter and More Oracle Database Access C API describe the OCI relational function calls and cover the functions in the basic OCI.

> **See Also:**
>
> Error Handling in OCI for information about return codes and error handling

### 25.1.1 Conventions for OCI Functions

Introduces the conventions used for OCI functions.

For each function, the following information is listed:

### 25.1.2 Purpose

What is the functions purpose?

A brief description of the action performed by the function.

## 25.1.3 Syntax

What is the function declaration?

The function declaration.

## 25.1.4 Parameters

A description of each of the function's parameters and their modes.

This description includes the parameter's mode. The mode of a parameter has three possible values, as described in Table 25-1.

**Table 25-1 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 |

## 25.1.5 Comments

Describes more detailed information about the function (if available).

This information may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

## 25.1.6 Returns

This optional section describes the possible values that can be returned.

This section can be found either before or after the Comments section.

## 25.1.7 Example

Provides acomplete or partial code example demonstrating the use of the function call being described.

Not all function descriptions include an example.

## 25.1.8 Related Functions

Shows a list of related function calls to the function if applicable.

A list of related function calls.

## 25.1.9 About Calling OCI Functions

Provides useful information about calling OCI functions.

Unlike earlier versions of 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, with the following exceptions:

- 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

### 25.1.10 Server Round-Trips for LOB Functions

Indicates the number of server round-trips required for individual OCI LOB functions.

For a table showing the number of server round-trips required for individual OCI LOB functions, see OCI Function Server Round-Trips.

## 25.2 Connect, Authorize, and Initialize Functions

Lists and describes connect, authorize, and initialize functions.

Table 25-1 describes the OCI connect, authorize, and initialize functions that are described in this section.

**Table 25-2 Connect, Authorize, and Initialize Functions**

| Function | Purpose |
|---|---|
| OCIAppCtxClearAll() | Clear all attribute-value information in a namespace of an application context |
| OCIAppCtxSet() | Set an attribute and its associated value in a namespace of an application context |
| OCIConnectionPoolCreate() | Initialize the connection pool |
| OCIConnectionPoolDestroy() | Destroy the connection pool |
| OCIDBShutdown() | Shut down Oracle Database |
| OCIDBStartup() | Start an Oracle Database instance |
| OCIEnvCreate() | Create and initialize an OCI environment handle |
| OCIEnvNlsCreate() | Create and initialize an environment handle for OCI functions to work under. Enable you to set character set ID and national character set ID at environment creation time. |
| OCILogoff() | Release a session that was retrieved using `OCILogon2()` or `OCILogon()` |
| OCILogon() | Simplify single-session logon |
| OCILogon2() | Create a logon session in various modes |
| OCIRequestDisableReplay() | Disables application continuity replay for a session for the duration of the current database request. |

**Table 25-2 (Cont.) Connect, Authorize, and Initialize Functions**

| Function | Purpose |
|---|---|
| OCIServerAttach() | Attach to a server; initialize server context handle |
| OCIServerDetach() | Detach from a server; uninitialize server context handle |
| OCISessionBegin() | Authenticate a user |
| OCISessionEnd() | Terminate a user session |
| OCISessionGet() | Get a session from a session pool |
| OCISessionPoolCreate() | Initialize a session pool |
| OCISessionPoolDestroy() | Destroy a session pool |
| OCISessionRelease() | Release a session |
| OCITerminate() | Detach from a shared memory subsystem |

## 25.2.1 OCIAppCtxClearAll()

Clears all attribute-value information in a namespace of an application context.

### Purpose

Clears all attribute-value information in a namespace of an application context.

### Syntax

```
sword OCIAppCtxClearAll ( void      *sesshndl,
                          void      *nsptr,
                          ub4        nsptrlen,
                          OCIError  *errhp,
                          ub4        mode ;
```

### Parameters

**sesshndl (IN/OUT)**
Pointer to a session handle.

**nsptr (IN)**
Pointer to the namespace string (currently only `CLIENTCONTEXT`).

**nsptrlen (IN)**
Length of the namespace string.

**errhp (OUT)**
An error handle that can be passed to `OCIErrorGet()`.

**mode (IN)**
Mode (`OCI_DEFAULT` is the default).

### Returns

Returns an error number.

### Comments

This cleans up the context information on the server side during the next call to the server. This namespace information is cleared from the session handle after the information has been sent to the server and must be set up again if needed.

### Related Topics

- OCIAppCtxSet()
  Sets an attribute and its associated value in a namespace of an application context.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.2 OCIAppCtxSet()

Sets an attribute and its associated value in a namespace of an application context.

### Purpose

Sets an attribute and its associated value in a namespace of an application context.

### Syntax

```
sword OCIAppCtxSet ( void      *sesshndl,
                     void      *nsptr,
                     ub4        nsptrlen,
                     void      *attrptr,
                     ub4        attrptrlen,
                     void      *valueptr,
                     ub4        valueptrlen,
                     OCIError  *errhp,
                     ub4        mode );
```

### Parameters

**sesshndl (IN/OUT)**
Pointer to a session handle.

**nsptr (IN)**
Pointer to the namespace string (currently only `CLIENTCONTEXT`).

**nsptrlen (IN)**
Length of the namespace string.

**attrptr (IN)**
Pointer to the attribute string.

**attrptrlen (IN)**
The length of the string pointed to by `attrptr`.

**valueptr (IN)**
Pointer to the value string.

**valueptrlen (IN)**
The length of the string pointed to by `valueptr`.

**errhp (OUT)**
An error handle that can be passed to `OCIErrorGet()`.

**mode (IN)**
Mode (`OCI_DEFAULT` is the default).

### Returns

Returns an 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 after the information has been sent to the server and must be set up again if needed.

**Related Topics**

- OCIAppCtxClearAll()
  Clears all attribute-value information in a namespace of an application context.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.3 OCIConnectionPoolCreate()

Initializes the connection pool.

### 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 that 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 connections are opened to the server by `OCIConnectionPoolCreate()`. After the connection pool is created, connections are opened only when necessary.
Generally, this parameter should be set to the number of concurrent statements that the application is planning or expecting to run.

**connMax (IN)**
Specifies the maximum number of connections that can be opened to the database. After 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 is less than `connMax`. Valid values are 0 and higher.

**poolUsername (IN)**
Connection pooling requires an implicit primary session. 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()` is called with `mode` set to `OCI_DEFAULT`.

To change the pool attributes dynamically (for example, to 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` contain values to be used in subsequent `OCIServerAttach()` and `OCILogon2()` calls in place of the database name and the database name length arguments.

**Related Topics**

- OCIConnectionPoolDestroy()
  Destroys the connection pool.
- OCILogon2()
  Gets a session.
- OCIServerAttach()
  Creates an access path to a data source for OCI operations.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Connection Pool Handle Attributes
  Lists and describes connection pool handle attributes.

## 25.2.4 OCIConnectionPoolDestroy()

Destroys the connection pool.

**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 that can be passed to `OCIErrorGet()`.

**mode (IN)**
Currently, this function supports only the `OCI_DEFAULT` mode.

**Related Topics**

- OCIConnectionPoolCreate()
  Initializes the connection pool.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.5 OCIDBShutdown()

Shuts down an Oracle Database instance.

### 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 that can be passed to `OCIErrorGet()` for diagnostic information when there is 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; for example, 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 through a dispatcher. When shutting down in any mode other than `OCI_DBSHUTDOWN_ABORT`, use the following procedure:

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.

**Related Topics**

- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIDBStartup()
  Starts an Oracle Database instance.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Database Startup and Shutdown
  This chapter describes topics about OCI database startup and shutdown.

## 25.2.6 OCIDBStartup()

Starts an Oracle Database instance.

### 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 that can be passed to `OCIErrorGet()` for diagnostic information when there is 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 the instance, but does not mount or open the database. Same as `STARTUP NOMOUNT`.

**flags (IN)**
`OCI_DEFAULT` - Allows database access to all users.
`OCI_DBSTARTUPFLAG_RESTRICT` - Allows database access only to users with both the `CREATE SESSION` and `RESTRICTED SESSION` privileges (normally, the DBA).

`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.

### 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 through 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`) is used. A call to `OCIDBStartup()` starts one instance on the server.

### Related Topics

- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIDBShutdown()
  Shuts down an Oracle Database instance.
- OCIServerAttach()
  Creates an access path to a data source for OCI operations.
- OCISessionBegin()
  Creates a user session and begins a user session for a given server.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Database Startup and Shutdown
  This chapter describes topics about OCI database startup and shutdown.

## 25.2.7 OCIEnvCreate()

Creates and initializes an environment handle for OCI functions to work under.

### Purpose

Creates and initializes an environment handle for OCI functions to work under.

### Syntax

```
sword OCIEnvCreate   ( OCIEnv       **envhpp,
                       ub4            mode,
                       const void    *ctxp,
                       const void    *(*malocfp)
                                      (void  *ctxp,
                                      size_t size),
                       const void    *(*ralocfp)
                                      (void  *ctxp,
                                      void  *memptr,
                                      size_t newsize),
                       const void     (*mfreefp)
                                      (void  *ctxp,
                                      void  *memptr))
                       size_t         xtramemsz,
                       void         **usrmempp );
```

### Parameters

**envhpp (OUT)**
A pointer to an environment handle whose encoding setting is specified by `mode`. The setting is 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` - Uses publish-subscribe notifications.
- `OCI_NO_UCB` - Suppresses the calling of the dynamic callback routine `OCIEnvCallback()`. The default behavior is to allow calling of `OCIEnvCallback()` when the environment is created.

  > **See Also:**
  >
  > "Dynamic Callback Registrations"

- `OCI_ENV_NO_MUTEX` - No mutual exclusion (mutex) locking occurs in this mode. All OCI calls done on the environment handle, or on handles derived from the environment handle, must be serialized. `OCI_THREADED` must also be specified when `OCI_ENV_NO_MUTEX` is specified.
- `OCI_SUPPRESS_NLS_VALIDATION` - Suppresses NLS character validation; NLS character validation suppression is on by default beginning with Oracle Database 11*g* Release 1 (11.1). Use `OCI_ENABLE_NLS_VALIDATION` to enable NLS character validation. See Comments for more information.
- `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 nor `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 is turned on; otherwise it is turned off, which is the default setting in OCI.
- `OCI_ENABLE_NLS_VALIDATION` - Enables NLS character validation. See Comments for more information.

**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 reallocation 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.

**memptr (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 the 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()` call.

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 you want 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, the `OCIStmtPrepare2()` function performs the N' substitution on the SQL text and stores 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, is 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 performed.

If you are writing a DLL or a shared library using the OCI library, then use this call instead of the deprecated `OCIInitialize()` call.

> **See Also:**
>
> "User Memory Allocation" for more information about the `xtramemsz` parameter and user memory allocation

Regarding `OCI_SUPPRESS_NLS_VALIDATION` and `OCI_ENABLE_NLS_VALIDATION` modes, by default, when client and server character sets are identical, and client and server releases are both Oracle Database 11*g* Release 1 (11.1) or later, OCI does not validate character data in the interest of better performance. This means that if the application inserts a character string with partial multibyte characters (for example, at the end of a bind variable), then such strings could get persisted in the database as is.

Note that if either the client or the server release is older than Oracle Database 11*g* Release 1 (11.1), then OCI does not allow partial characters.

The `OCI_ENABLE_NLS_VALIDATION` mode, which was the default until Oracle Database 10*g* Release 2 (10.2), ensures that partial multibyte characters are not persisted in the database (when client and server character sets are identical). If the application can produce partial multibyte characters, and if the application can run in an environment where the client and server character sets are identical, then Oracle recommends using the `OCI_ENABLE_NLS_VALIDATION` mode explicitly in order to ensure that such partial characters get stripped out.

#### Example

Creating a Thread-Safe OCI Environment with N' Substitution Turned On

```
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,
         (void  *)0, (void  * (*)(void  *, size_t))0,
         (void  * (*)(void  *, void  *, size_t))0,
         (void (*)(void  *, void  *))0,
         (size_t)0, (void  **)0))
{
  printf("Failed: OCIEnvCreate()\n");
  return 1;
}
...
```

**Related Topics**

- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIHandleFree()
  Explicitly deallocates a handle
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCITerminate()
  Detaches the process from the shared memory subsystem and releases the shared memory.
- OCIStmtPrepare2()
  Prepares a SQL or PL/SQL statement for execution.

## 25.2.8 OCIEnvNlsCreate()

Creates and initializes an environment handle for OCI functions to work under.

**Purpose**

This function is an enhanced version of the `OCIEnvCreate()` function.

**Syntax**

```
sword OCIEnvNlsCreate   ( OCIEnv        **envhpp,
                          ub4             mode,
                          void           *ctxp,
                          void          *(*malocfp)
                                             (void  *ctxp,
                                              size_t size),
                          void          *(*ralocfp)
                                             (void  *ctxp,
                                              void  *memptr,
                                              size_t newsize),
                          void          (*mfreefp)
                                             (void  *ctxp,
                                              void  *memptr))
                          size_t          xtramemsz,
                          void          **usrmempp
                          ub2             charset,
                          ub2             ncharset );
```

**Parameters**

**envhpp (OUT)**
A pointer to an environment handle whose encoding setting is specified by `mode`. The setting is 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` - Uses publish-subscribe notifications.
- `OCI_NO_UCB` - Suppresses the calling of the dynamic callback routine `OCIEnvCallback()`. The default behavior is to allow calling of `OCIEnvCallback()` when the environment is created.

> **See Also:**
>
> Dynamic Callback Registrations

- `OCI_ENV_NO_MUTEX` - No mutual exclusion (mutex) locking occurs in this mode. All OCI calls done on the environment handle, or on handles derived from the environment handle, must be serialized. `OCI_THREADED` must also be specified when `OCI_ENV_NO_MUTEX` is specified.
- `OCI_SUPPRESS_NLS_VALIDATION` - Suppresses NLS character validation; NLS character validation suppression is on by default beginning with Oracle Database 11*g* Release 1 (11.1). Use `OCI_ENABLE_NLS_VALIDATION` to enable NLS character validation. See Comments for more information.
- `OCI_NCHAR_LITERAL_REPLACE_ON` - Turns on N' substitution.
- `OCI_NCHAR_LITERAL_REPLACE_OFF` - Turns off N' substitution. If neither this mode nor `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 is turned on; otherwise it is turned off, the default setting in OCI.
- `OCI_ENABLE_NLS_VALIDATION` - Enables NLS character validation. See Comments for more information.

**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 reallocation 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.

**memptr (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 the 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 you specify.

After you use `OCIEnvNlsCreate()` to create the environment handle, the actual lengths and returned lengths of bind and define handles are always expressed in number of bytes. This applies to the following calls:

- `OCIBindByName()` and `OCIBindByName2()`
- `OCIBindByPos()` and `OCIBindByPos2()`
- `OCIBindDynamic()`
- `OCIDefineByPos()` and `OCIDefineByPos2()`
- `OCIDefineDynamic()`

This function enables you to set `charset` and `ncharset` IDs at environment creation time. It is an enhanced version of the `OCIEnvCreate()` function.

`OCIEnvNlsCreate()` must be called with `charset` and `ncharset` either both zero or both non-zero. Failure to do so results in an error (`ORA-24820`).

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, the function 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 you want 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, the `OCIStmtPrepare2()` function performs the N' substitution on the SQL text and stores 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, is 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 performed.

If you are writing a DLL or a shared library using the OCI library, then use this call instead of the deprecated `OCIInitialize()` call.

**See Also:**

- User Memory Allocation for more information about the `xtramemsz` parameter and user memory allocation
- `OCIEnvCreate()` for a code example illustrating setting N' substitution in a related function

Regarding `OCI_SUPPRESS_NLS_VALIDATION` and `OCI_ENABLE_NLS_VALIDATION` modes, by default, when client and server character sets are identical, and client and server releases are both Oracle Database 11*g* Release 1 (11.1) or later, OCI does not validate character data in the interest of better performance. This means that if the application inserts a character string with partial multibyte characters (for example, at the end of a bind variable), then such strings could get persisted in the database as is.

Note that if either the client or the server release is older than Oracle Database 11*g* Release 1 (11.1), then OCI does not allow partial characters.

The `OCI_ENABLE_NLS_VALIDATION` mode, which was the default until Oracle Database 10*g* Release 2 (10.2), ensures that partial multibyte characters are not persisted in the database (when client and server character sets are identical). If the application can produce partial multibyte characters, and if the application can run in an environment where the client and server character sets are identical, then Oracle recommends using the `OCI_ENABLE_NLS_VALIDATION` mode explicitly in order to ensure that such partial characters get stripped out.

**Related Topics**

- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIHandleFree()
  Explicitly deallocates a handle
- OCITerminate()
  Detaches the process from the shared memory subsystem and releases the shared memory.
- OCINlsEnvironmentVariableGet()
  Returns the character set ID from `NLS_LANG` or the national character set ID from `NLS_NCHAR`.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIStmtPrepare2()
  Prepares a SQL or PL/SQL statement for execution.

## 25.2.9 OCILogoff()

Releases a session that was retrieved using `OCILogon2()` or `OCILogon()`.

**Purpose**

Releases a session that was retrieved using `OCILogon2()` or `OCILogon()`.

**Syntax**

```
sword OCILogoff ( OCISvcCtx      *svchp
                  OCIError       *errhp );
```

**Parameters**

**svchp (IN)**
The service context handle that was used in the call to `OCILogon()` or `OCILogon2()`.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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, this function closes the session or connection. For connection pooling, it closes the session and returns the connection to the pool. For session pooling, it returns the session or connection pair to the pool.

**Related Topics**

- OCILogon()
  Creates a simple logon session.
- OCILogon2()
  Gets a session.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> "Application Initialization, Connection, and Session Creation" for more information about logging on and off in an application

## 25.2.10 OCILogon()

Creates a simple logon session.

**Purpose**

Creates 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 `username`, 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 see "Application Initialization, Connection, and Session Creation".

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 `OCIArrayDescriptorAlloc()` on the service context handle.

### Related Topics

- OCILogoff()
  Releases a session that was retrieved using `OCILogon2()` or `OCILogon()`.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIArrayDescriptorAlloc()
  Allocates an array of descriptors.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.11 OCILogon2()

Gets a session.

**Purpose**

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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**svchp (IN/OUT)**
Address of an OCI service context pointer. This is filled with a server and session handle.
In the default case, a new session and server handle is allocated, the connection and session are started, and the service context is populated with these handles.
For connection pooling, a new session handle is allocated, and the session is started over a virtual connection from the connection pool.
For session pooling, the service context is populated with an existing session or server handle pair from the session pool.
Note that you must not change any attributes of the server and user or session handles associated with the service context pointer. Doing so results 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.
For connection pooling, this indicates the connection pool from which to retrieve the virtual connection to start 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.
The `dbname` 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 (nonpooling 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 to use connection pooling.

`OCI_LOGON2_STMTCACHE` - Enable statement caching.

To use proxy authentication for connection pooling, the password must be set to `NULL`. You are then 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 to use session pooling.

`OCI_LOGON2_STMTCACHE` - Enable statement caching.

`OCI_LOGON2_PROXY` - Use proxy authentication. You are 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 Topics**

- OCILogon()
  Creates a simple logon session.
- OCILogoff()
  Releases a session that was retrieved using `OCILogon2()` or `OCILogon()`.
- OCISessionGet()
  Gets a session.
- OCISessionRelease()
  Releases a session that was retrieved using `OCISessionGet()`.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIConnectionPoolCreate()
  Initializes the connection pool.
- OCISessionPoolCreate()
  Initializes a session pool for use with OCI session pooling and database resident connection pooling (DRCP).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.12 OCIRequestDisableReplay()

Disables application continuity replay for a session for the duration of the current database request.

**Purpose**

Triggers OCI to stop recording calls until the end of an application request (the end of a request is marked by `OCISessionRelease()`). Use this call when Application Continuity for OCI is enabled but the application is entering a section of code that is not replayable.

**Syntax**

```
OCIRequestDisableReplay(OCISvcCtx *svchp,
                        OCIError  *errhp,
                        ub4        mode);
```

### Parameters

**svchp (IN/OUT)**
The service context; the service context handle must be initialized and have a session handle associated with it.

**errhp (IN/OUT)**
The error handle.

**mode (IN)**
This call supports the following modes:

- `OCI_DEFAULT`

### Usage Notes

See OCI and Application Continuity for more information. See *Oracle Real Application Clusters Administration and Deployment Guide* for information about enabling Application Continuity.

## 25.2.13 OCIServerAttach()

Creates an access path to a data source for OCI operations.

### Purpose

Creates an access path to a data source for OCI operations.

### Syntax

```
sword OCIServerAttach ( OCIServer     *srvhp,
                        OCIError      *errhp,
                        const OraText *dblink,
                        sb4            dblink_len,
                        ub4            mode );
```

### Parameters

**srvhp (IN/OUT)**
An uninitialized server handle, which is initialized by this call. Passing in an initialized server handle causes an error.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**dblink (IN)**
Specifies the database server to use. This parameter points to a character string that 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` encoding mode or not, depending on the `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.

Because 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 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()`. After 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 Database shadow process is started. `OCISessionEnd()` and `OCIServerDetach()` should be called to clean up the Oracle Database shadow process. Otherwise, the shadow processes accumulate and cause the Linux or UNIX system to run out of processes. If the database is restarted and there are not enough processes, the database may not start up.

See OCI and Application Continuity for more information about Application Continuity. See *Oracle Real Application Clusters Administration and Deployment Guide* for information about enabling Application Continuity.

### Example

The following code 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.

Using the OCIServerAttach() Call

```
OCIHandleAlloc( (void  *) envhp, (void  **) &srvhp, (ub4)
     OCI_HTYPE_SERVER, 0, (void  **) 0);
OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
```

```
OCIHandleAlloc( (void  *) envhp, (void  **) &svchp, (ub4)
     OCI_HTYPE_SVCCTX, 0, (void  **) 0);
/* set attribute server context in the service context */
OCIAttrSet( (void  *) svchp, (ub4) OCI_HTYPE_SVCCTX, (void  *) srvhp,
     (ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);
```

**Related Topics**

- OCIServerDetach()
  Deletes an access path to a data source for OCI operations.
- OCIConnectionPoolCreate()
  Initializes the connection pool.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCISessionEnd()
  Terminates a user session context created by `OCISessionBegin()`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.14 OCIServerDetach()

Deletes an access path to a data source for OCI operations.

**Purpose**

Deletes an access path 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 is reset to an uninitialized state. The handle is not deallocated.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 path a to data source for OCI operations. The access path was established by a call to `OCIServerAttach()`.

**Related Topics**

- OCIServerAttach()
  Creates an access path to a data source for OCI operations.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.15 OCISessionBegin()

Creates a user session and begins a user session for a given server.

**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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**usrhp (IN/OUT)**
A handle to a user session context, which is initialized by this call.

**credt (IN)**
Specifies the type of credentials to use for establishing the user session. Valid values for `credt` are:

- `OCI_CRED_RDBMS` - Authenticate using a database 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 can only ever be set with the 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 can 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 nonmigratable user session, which becomes the "creator" session of the migratable session. That is, a migratable session must have a nonmigratable 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, you are authenticated for SYSDBA access.
- `OCI_SYSOPER` - In this mode, you are authenticated for SYSOPER access.
- `OCI_SYSASM` - In this mode, you are authenticated for SYSASM access.
- `OCI_SYSBKB` - In this mode, you are authenticated for SYSBACKUP access.
- `OCI_SYSDGD` - In this mode, you are authenticated for SYSDG access.
- `OCI_SYSKMT` - In this mode, you are authenticated for SYSKM access.
- `OCI_SYSRAC` - In this mode, you are authenticated for SYSRAC access.
- `OCI_PRELIM_AUTH` - This mode can only be used with `OCI_SYSDBA` or `OCI_SYSOPER` to authenticate for certain administration tasks.
- `OCI_STMT_CACHE` - Enables statement caching with default size on the given service handle. It is optional to pass this mode if the application is going to explicitly set the size later using `OCI_ATTR_STMTCACHESIZE` on that service handle.

**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 database 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.

If Application Continuity for OCI is enabled, `OCISessionBegin()` implicitly marks the beginning of an application request. When using the OCI session pool APIs, it is not necessary to call `OCIRequestBegin()` and `OCIRequestEnd()`.

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 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 `void` pointers. The string buffers then are 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 be associated with a nonmigratable user handle. The user ID of that user handle becomes the ownership ID of the migratable user session. Every migratable session must have a nonmigratable 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 the `OCI_MIGRATE` mode is specified, then the user authentication can be set with different server handles. However, the user session context can only be used with server handles that 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 nonmigratable 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 the `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 Database supports passing this flag only for compatibility reasons. Do not use the `OCI_MIGRATE` flag, as the perception that you get when using a connection pool is of sessions having their own dedicated (virtual) connections that are transparently multiplexed onto real connections.

`OCI_SYSDBA`, `OCI_SYSOPER`, and `OCI_PRELIM_AUTH` can only be used with a primary user session context.

To provide credentials for a call to `OCISessionBegin()`, two methods are supported. The first method 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 reused in a future call to `OCISessionBegin()`. Otherwise, they must be reset to new values before the next `OCISessionBegin()` call.

The second method is to use 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 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 code 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.

Using the OCISessionBegin() Call

```
/* allocate a user session handle */
OCIHandleAlloc((void  *)envhp, (void  **)&usrhp, (ub4)
     OCI_HTYPE_SESSION, (size_t) 0, (void  **) 0);
OCIAttrSet((void  *)usrhp, (ub4)OCI_HTYPE_SESSION, (void  *)"hr",
     (ub4)strlen("hr"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((void  *)usrhp, (ub4)OCI_HTYPE_SESSION, (void  *)"hr",
     (ub4)strlen("hr"), OCI_ATTR_PASSWORD, errhp);
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
     OCI_DEFAULT));
OCIAttrSet((void  *)svchp, (ub4)OCI_HTYPE_SVCCTX, (void  *)usrhp,
     (ub4)0, OCI_ATTR_SESSION, errhp);
```

**Related Topics**

- OCISessionEnd()
  Terminates a user session context created by `OCISessionBegin()`.
- OCIServerAttach()
  Creates an access path to a data source for OCI operations.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIAttrGet()
  Gets the value of an attribute of a handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.16 OCISessionEnd()

Terminates a user session context created by `OCISessionBegin()`.

**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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**usrhp (IN)**
Deauthenticate this user. If this parameter is passed as `NULL`, the user in the service context handle is deauthenticated.

**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 it is not being used. Resources allocated on the server for this user are freed. The user session handle can be reused in a new call to `OCISessionBegin()`.

If Application Continuity for OCI is enabled, `OCISessionEnd()` implicitly marks the end of an application request. OCI stops recording calls and purges its call history. When using the OCI session pool APIs, it is not necessary to call `OCIRequestBegin()` and `OCIRequestEnd()`.

### Related Topics

- OCISessionBegin()
  Creates a user session and begins a user session for a given server.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.17 OCISessionGet()

Gets a session.

### Purpose

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, the environment handle in which the respective pool was created.

**errhp (IN/OUT)**
OCI error handle. Multithreaded OCI clients should pass a unique `errhp` to the `OCISessionGet()` call.

**svchp (OUT)**
Address of an OCI service context pointer. This is filled with a server and session handle.
In the default case, a new session and server handle are allocated, the connection and session are started, and the service context is populated with these handles.
For connection pooling, a new session handle is allocated, and the session is started over a virtual connection from the connection pool.
For session pooling, the service context is 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 results 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 other words, setting the mode to `OCI_SPC_HOMOGENEOUS`, the authentication information is not used, while all other possible specified attributes are used.
The attributes that can be set on the `OCIAuthInfo` handle can be categorized into pre-session-creation attributes and post-session-creation attributes. The pre-session-creation attributes are:
**Pre-session-creation attributes**
Pre-session-creation attributes are those OCI attributes that must be specified before a session is created. These attributes are used to create a session and cannot be changed after a session is created. The pre-session creation attributes are:
`OCI_ATTR_USERNAME`
`OCI_ATTR_PASSWORD`
`OCI_ATTR_CONNECTION_CLASS`
`OCI_ATTR_PURITY`
`OCI_ATTR_DISTINGUISHED_NAME`
`OCI_ATTR_CERTIFICATE`
`OCI_ATTR_INITIAL_CLIENT_ROLES`
`OCI_ATTR_APPCTX_SIZE`
`OCI_ATTR_EDITION`
`OCI_ATTR_DRIVER_NAME`

**Post-session-creation attributes**
Post-session-creation attributes are those that can be specified after a session is created. They can be changed freely after a session is created as many times as desired. The following attributes can be set on the `OCISession` handle after the session has been created:
`OCI_ATTR_CLIENT_IDENTIFIER`
`OCI_ATTR_CURRENT_SCHEMA`
`OCI_ATTR_MODULE`
`OCI_ATTR_ACTION`
`OCI_ATTR_DBOP`
`OCI_ATTR_CLIENT_INFO`
`OCI_ATTR_COLLECT_CALL_TIME`
`OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE`
`OCI_ATTR_SESSION_STATE`

> **See Also:**
>
> - Session Pooling in OCI
> - `cdemosp.c` in the `demo` directory
> - User Session Handle Attributes for more information about the attributes
> - The Comments section

**dbName (IN)**
For the default case, this indicates the connect string to use to connect to the Oracle database.
For connection pooling, it indicates the connection pool to retrieve the virtual connection from, to start 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 used only for session pooling.
This indicates the type of session that the user wants. If you want a default session, you must set this to `NULL`. See the Comments for a detailed explanation of this parameter.
Beginning with 12*c* Release 2 (12.2), a tag can have multiple properties. This is referred to as a multi-property tag. A multi-property tag is comprised of one or more <property-name>=<property-value> pairs separated by a semi-colon, where <property-name>=<property-value> are both strings. See About Using Tags in Session Pools for more information.

**tagInfo_len (IN)**
The length, in bytes, of `tagInfo`. Used for session pooling only.

**retTagInfo (OUT)**
This parameter is used only for session pooling. This indicates the type of session that is returned to the user. See the Comments for a detailed explanation of this parameter.

**retTagInfo_len (OUT)**
The length, in bytes, of `retTagInfo`. Used for session pooling only.

**found (OUT)**
This parameter is used only 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`. Otherwise, `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` - Supported only for heterogeneous pools.
- `OCI_SESSGET_CUSTOM_POOL`
- `OCI_SESSGET_MULTIPROPERTY_TAG`
- `OCI_SESSGET_PURITY_NEW`
- `OCI_SESSGET_PURITY_SELF`
- `OCI_SESSGET_SPOOL_MATCHANY`
- `OCI_SESSGET_STMTCACHE`
- `OCI_SESSGET_SYSDBA`

In the default (nonpooling) case, the following modes are valid:

`OCI_SESSGET_STMTCACHE` - Enables statement caching in the session.

`OCI_SESSGET_CREDEXT` - Returns a session authenticated with external credentials.

`OCI_SESSGET_SYSDBA` - Returns a session with `SYSDBA` privilege for either nonpooling or for session pooling.

For connection pooling, the following modes are valid:

`OCI_SESSGET_CPOOL` - Must be set to use connection pooling.

`OCI_SESSGET_STMTCACHE` - Enables statement caching in the session.

`OCI_SESSGET_CREDPROXY` - Returns 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` - Returns a session authenticated with external credentials.

For session pooling, the following modes are valid:

`OCI_SESSGET_SPOOL` - Must be set to use session pooling.

`OCI_SESSGET_SYSDBA` - Returns a session with `SYSDBA` privilege for either nonpooling or for session pooling.

`OCI_SESSGET_CREDEXT` - Returns a session authenticated with external credentials.

`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` - Refers to the tagging behavior. If this mode is set, then a session that has a different tag than what was asked for, may be returned. See the Comments section.

`OCI_SESSGET_MULTIPROPERTY_TAG` - Must be set when using multiple properties in case applications previously used a (;) semi-colon character was in a tag name; the semi-colon character is used as a separator for specifying multiple properties. See the Comments section.

For database resident connection pooling, the following modes are valid:

`OCI_SESSGET_MULTIPROPERTY_TAG` - Must be set when using multiple properties in case applications used a (;) semi-colon character was in a tag name; the semi-colon character is used as a separator for specifying multiple properties. See the Comments section.

`OCI_SESSGET_PURITY_SELF` - The application can use a session that has been used before. You can also specify application-specific tags.

`OCI_SESSGET_PURITY_NEW` - The application requires a new session that is not tainted with any prior session state. This is the default.

For custom pools, the following modes are valid:

`OCI_SESSGET_CUSTOM_POOL` - Must be set by applications that are not using OCI session pool, but instead are using custom pools. This attribute is set to support shard key lookup in custom pools.

**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 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) is 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 is returned. If `OCI_SESSGET_SPOOL_MATCHANY` is not set, then a session with a different tag is never returned.

The following code example demonstrates the use of `OCI_ATTR_MODULE` with session pooling.

Using the OCI_ATTR_MODULE Attribute with OCI Session Pooling

```
Oratext *module = (Oratext*) "mymodule";
/* Allocate the pool handle */
 checkerr(errhp,OCIHandleAlloc(envhp,(void**)&poolhp,
                             OCI_HTYPE_SPOOL,0,0));

 checkerr(errhp,OCISessionPoolCreate(envhp,
                             errhp,poolhp,&poolname,&pnamelen,
                             (oratext*)conn_str,
                             len,min,max,incr,0,0,0,0,OCI_DEFAULT));

 /* Allocate the auth handle for session get */
 checkerr(errhp, OCIHandleAlloc(envhp,
                 (void**)&authp, OCI_HTYPE_AUTHINFO, 0,0));

 checkerr(errhp,OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
                  username, strlen((char*)username), OCI_ATTR_USERNAME,errhp);
 checkerr(errhp,OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
                  password, strlen((char*)password), OCI_ATTR_PASSWORD,
                  errhp));

 checkerr(errhp,OCISessionGet(envhp,errhp,
              &svchp,authp,poolname, pnamelen,0,0,0,0,0,
              OCI_SESSGET_SPOOL));

/* Get the user handle from the service context handle */
checkerr(errhp, OCIAttrGet(svhcp, OCI_HTYPE_SVCCTX, &usrhp_svc,
               0,OCI_ATTR_SESSION,errhp));

/* Set module name on the user handle that you obtained */
checkerr (errhp, OCIAttrSet(usrhp_svc, OCI_HTYPE_SESSION, module,
               strlen((char*)module), OCI_ATTR_MODULE,errhp));
/* Make Database calls. */
```

### Restrictions on Attributes Supported for OCI Session Pools

You can use the following pre-session-creation attributes with OCI session pools:

```
OCI_ATTR_EDITION
OCI_ATTR_DRIVER_NAME
OCI_ATTR_USERNAME,
OCI_ATTR_PASSWORD,
OCI_ATTR_CONNECTION_CLASS,
OCI_ATTR_PURITY
```

However, `OCI_ATTR_EDITION` and `OCI_ATTR_DRIVERNAME` can only be specified during `OCISessionPoolCreate()` by setting them on the `OCIAuthInfo` handle that is an attribute of `OCISPool` handle. They cannot be specified on the `OCIAuthInfo` handle passed into individual `OCISessionGet()` calls. This ensures that all sessions that are part of an OCI session pool have uniform values for these attributes.

The following code example shows how to use the `OCI_ATTR_EDITION` attribute with an OCI session pool.

### Using the OCI_ATTR_EDITION Attribute with OCI Session Pooling

```
/* allocate the auth handle to be set on the spool handle */
  checkerr(errhp, OCIHandleAlloc(envhp,(void**)&authp_sp,
                  OCI_HTYPE_AUTHINFO, 0,0));

  /* Set the edition on the auth handle */
  checkerr(errhp,OCIAttrSet(authp_sp, OCI_HTYPE_AUTHINFO,
                   "Patch_Bug_12345", strlen("Patch_Bug_12345"),
                  OCI_ATTR_EDITION,errhp));

/* Allocate the pool handle */
 checkerr(errhp,OCIHandleAlloc(envhp,(void**)&poolhp,
                             OCI_HTYPE_SPOOL,0,0));

  /* Set the auth handle created above on the spool handle */
 checkerr(errhp,OCIAttrSet(poolhp, OCI_HTYPE_SPOOL,authp_sp,
                  0,OCI_ATTR_SPOOL_AUTH,errhp));
checkerr(errhp,OCISessionPoolCreate(envhp,
                             errhp,poolhp,&poolname,&pnamelen,
                             (oratext*)conn_str,
                             len,min,max,incr,0,0,0,0,OCI_DEFAULT));

  /* Allocate the auth handle for session get */
  checkerr(errhp, OCIHandleAlloc(envhp,
                 (void**)&authp_sessget, OCI_HTYPE_AUTHINFO, 0,0));

  checkerr(errhp,OCIAttrSet(authp_sessget, OCI_HTYPE_AUTHINFO,
                  username, strlen((char*)username), OCI_ATTR_USERNAME,errhp);
  checkerr(errhp,OCIAttrSet(authp_sessget, OCI_HTYPE_AUTHINFO,
                  password, strlen((char*)password), OCI_ATTR_PASSWORD,
                  errhp));

  checkerr(errhp,OCISessionGet(envhp,errhp,
              &svchp,authp_sessget,poolname, pnamelen,0,0,0,0,0,
              OCI_SESSGET_SPOOL));
```

You can use all post-session-creation attributes with OCI session pool. However, as a session pool can age out sessions, reuse preexisting sessions in the pool, or re-create new sessions transparently, Oracle recommends that the application explicitly set any post-session-creation attributes that it needs after getting a session from a pool. This ensures that the application logic works irrespective of the specific session returned by the OCI session pool.

When you specify the `OCI_SESSGET_MULTIPROPERTY_TAG`, the parameter `retTagInfo` needs to be interpreted in conjunction with the `found` parameter and the `mode` parameter values specified during `OCISessionGet()`. The following table specifies these semantics:

| Specified Modes | refTagInfo Parameter | found Parameter |
|---|---|---|
| `OCI_SESSGET_SPOOL \| OCI_SESSGET_MULTIPROPERTY_TAG` | All the properties set on the found session are returned as one composite string delimited by (;) semi-colon. | TRUE, only if the found session matches all requested properties and the session has no additional properties set.<br>FALSE, if no matching session was found. In this case, a new session, with no properties set, is returned. |

| Specified Modes | refTagInfo Parameter | found Parameter |
|---|---|---|
| `OCI_SESSGET_SPOOL \| OCI_SESSGET_MULTIPROPERTY_TAG \| OCISESSGET_SPOOL_MATCHANY` | All the properties set on the found session are returned as one composite string delimited by (;) semi-colon. | TRUE, if the found session matches all requested properties (but may have additional properties).<br>FALSE, if the found session did not match at least one property. |

There is a difference in the manner in which `retTagInfo` is returned in the case of multi-property tags as compared to non multi-property tags. In a non multi-property tag scenario,`retTagInfo` will be the same as the requested tag when the `found` parameter is returned as `TRUE`. Whereas in the case of a multi-property tag, the `retTagInfo` contains the complete set of properties associated with the returned session and further information should be deduced by the application from the `found` flag as shown in the previous table.

Note that `OCI_SESSGET_MULTIPROPERTY_TAG` (with `OCISessionGet()`) or `OCI_SESSRLS_MULTIPROPERTY_TAG` (with `OCISessionRelease()`) are only supported with OCI Session Pool (with or without DRCP). Thus, these modes cannot be used with standalone connections or with OCI Connection Pool.

**Related Topics**

- OCISessionRelease()
  Releases a session that was retrieved using `OCISessionGet()`.
- OCISessionPoolCreate()
  Initializes a session pool for use with OCI session pooling and database resident connection pooling (DRCP).
- OCISessionPoolDestroy()
  Destroys a session pool.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIConnectionPoolCreate()
  Initializes the connection pool.

## 25.2.18 OCISessionPoolCreate()

Initializes a session pool for use with OCI session pooling and database resident connection pooling (DRCP).

### Purpose

It starts `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 is to be created.

**errhp (IN/OUT)**
An error handle that 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 the sessions are started, sessions are opened only when necessary.
This value is used when `mode` is set to `OCI_SPC_HOMOGENEOUS`. Otherwise, it is ignored.

**sessMax (IN)**
Specifies the maximum number of sessions that can be opened in the session pool. After 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 is less than `sessMax`. The valid values are 0 and higher.

**userid (IN)**
Specifies the userid with which to start the sessions.

> **See Also:**
>
> Authentication Note in the Comments Section.

**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 want 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` are ignored.
- `OCI_SPC_STMTCACHE` - An OCI statement cache is created for the session pool. If the pool is not created with OCI statement caching turned on, server-side statement caching is automatically used. Note that in general, client-side statement caching gives better performance.

  > **See Also:**
  >
  > Statement Caching in OCI

- `OCI_SPC_HOMOGENEOUS` - All sessions in the pool are authenticated with the user name and password passed to `OCISessionPoolCreate()`. The authentication handle (parameter `authInfop`) 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. This mode can be used in database resident connection pooling (DRCP).
- `OCI_SPC_NO_RLB` - By default, the runtime connection load balancing is enabled in the session pool if the client and the server are capable of supporting it. To turn it off, use the new mode, `OCI_SPC_NO_RLB` mode of `OCISessionPoolCreate()`. You can only use this mode at the time of pool creation. If this mode is passed for a pool that has been created, an error, ORA-24411, is thrown.

### Comments

**Authentication Note**
A session pool can 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 about proxy connections, see
>
> - Client Access Through a Proxy
> - *Oracle Database SQL Language Reference*

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 that 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.

### Example

The following code example shows how to disable runtime load balancing.

Disabling Runtime Load Balancing

```
OCISessionPoolCreate(envhp, errhp, spoolhp, (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_SPC_HOMOGENEOUS | OCI_SPC_NO_RLB);
```

### Related Topics

- OCISessionRelease()
  Releases a session that was retrieved using `OCISessionGet()`.
- OCISessionGet()
  Gets a session.
- OCISessionPoolDestroy()
  Destroys a session pool.
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.2.19 OCISessionPoolDestroy()

Destroys a session pool.

### 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 that can be passed to `OCIErrorGet()`.

**mode (IN)**
Currently, `OCISessionPoolDestroy()` supports 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 are closed and the pool is destroyed. However, if this mode is not set, and there are busy sessions in the pool, an error is returned.

### Related Topics

- OCISessionPoolCreate()
  Initializes a session pool for use with OCI session pooling and database resident connection pooling (DRCP).
- OCISessionRelease()
  Releases a session that was retrieved using `OCISessionGet()`.
- OCISessionGet()
  Gets a session.

## 25.2.20 OCISessionRelease()

Releases a session that was retrieved using `OCISessionGet()`.

### Purpose

Releases 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 closes the session or connection. For connection pooling, it closes the session and returns the connection to the pool. For session pooling, it returns the session or connection pair to the pool, and any pending transaction is committed.

### 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 is closed.
In the connection pooling case, the session is closed and the connection released to the pool.
For session pooling, the session or connection pair associated with this service context is released to the pool.

**errhp (IN/OUT)**
The OCI error handle.

**tag (IN)**
This parameter is used only for session pooling.
This parameter is ignored unless mode `OCI_SESSRLS_RETAG` is specified. In this case, the session is labeled with this tag and returned to the pool. If this is `NULL`, then the session is not tagged.

**tag_len (IN)**
This parameter is used only 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_MULTIPROPERTY_TAG`
- `OCI_SESSRLS_RETAG`

You can only use `OCI_DEFAULT` for the default case and for connection pooling.

`OCI_SESSRLS_DROPSESS` and `OCI_SESSRLS_RETAG` are only used for session pooling.

When `OCI_SESSRLS_DROPSESS` is specified, the session is removed from the session pool.

Set `OCI_SESSRLS_MULTIPROPERTY_TAG`, if the tag is a multi-property tag. For more information, see Multi-Property Tags.

The tag on the session is altered if and only if `OCI_SESSRLS_RETAG` is set. If this mode is not set, the `tag` and `tag_len` parameters are ignored.

**Comments**

Be careful to pass in the correct tag when using the `tag` parameter. 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, however, 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 Topics**

- OCISessionGet()
  Gets a session.
- OCISessionPoolCreate()
  Initializes a session pool for use with OCI session pooling and database resident connection pooling (DRCP).
- OCISessionPoolDestroy()
  Destroys a session pool.
- OCILogon2()
  Gets a session.

## 25.2.21 OCITerminate()

Detaches the process from the shared memory subsystem and releases the shared memory.

**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 the `OCIEnvCreate()`, `OCIEnvNlsCreate()`, deprecated `OCIInitialize()` calls. The call tries 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 call `OCITerminate()` simultaneously, the fastest one releases the shared memory subsystem completely and the slower ones must terminate.

**Related Topics**

- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

# 25.3 Handle and Descriptor Functions

Lists and describes the OCI handle and descriptor functions.

Table 25-3 lists the OCI handle and descriptor functions that are described in this section.

**Table 25-3 Handle and Descriptor Functions**

| Function | Purpose |
|---|---|
| OCIArrayDescriptorAlloc() | Allocate an array of descriptors |
| OCIArrayDescriptorFree() | Free an array of descriptors |
| OCIAttrGet() | Get the value of an attribute of a handle |
| OCIAttrSet() | Set the value of an attribute of a handle or descriptor |
| OCIDescriptorAlloc() | Allocate and initialize a descriptor or LOB locator |
| OCIDescriptorFree() | Free a previously allocated descriptor |
| OCIHandleAlloc() | Allocate and initialize a handle |
| OCIHandleFree() | Free a previously allocated handle |
| OCIParamGet() | Get a parameter descriptor |
| OCIParamSet() | Set parameter descriptor in COR handle |

## 25.3.1 OCIArrayDescriptorAlloc()

Allocates an array of descriptors.

### Purpose

Allocates an array of descriptors.

### Syntax

```
sword OCIArrayDescriptorAlloc ( const void    *parenth,
                                void         **descpp,
                                const ub4      type,
                                ub4            array_size,
                                const size_t   xtramem_sz,
                                void         **usrmempp);
```

### Parameters

**parenth (IN)**
An environment handle.

**descpp (OUT)**
Returns a pointer to a contiguous array of descriptors of the desired type.

**type (IN)**
Specifies the type of descriptor or LOB locator to be allocated. For a list of types, see `OCIDescriptorAlloc()`.

**array_size (IN)**
Specifies the number of descriptors to allocate. An error is thrown when the call cannot allocate the number of descriptors.

**xtramem_sz (IN)**
Specifies an amount of user memory to be allocated for use by the application for the lifetime of the descriptors.

**usrmempp (OUT)**
Returns a pointer to an array of pointers numbering in `array_size`, since there is one chunk allocated for each of the descriptors allocated.

### Comments

This call returns `OCI_SUCCESS` if successful, or a suitable error if an out-of-memory condition occurs.

> **See Also:**
>
> User Memory Allocation for more information about the `xtramem_sz` parameter and user memory allocation

### Example

The following code example can be modified to allocate a large number of descriptors.

Allocating a Large Number of Descriptors

```
OCIDateTime *descpp1[500];
...
for (i = 0; i!=500; i++)
{
     /* Allocate descriptors */
OCIDescriptorAlloc((void  *)envhp,(void  **)&descpp1[i], OCI_DTYPE_TIMESTAMP_TZ,
                   0,(void  **)0);
}
...
```

The for loop in the previous code example can now be replaced with a single call, as shown in the following code example.

Allocating an Array of Descriptors

```
OCIArrayDescriptorAlloc((void *)envhp,(void **)&descpp1,
                        OCI_DTYPE_TIMESTAMP_TZ, 500, 0, (void **)0);
```

### Related Topics

- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.
- OCIArrayDescriptorFree()
  Free a previously allocated array of descriptors.

## 25.3.2 OCIArrayDescriptorFree()

Free a previously allocated array of descriptors.

### Purpose

Free a previously allocated array of descriptors.

### Syntax

```
sword OCIArrayDescriptorFree ( void       **descp,
                               const ub4    type );
```

### Parameters

**descp (IN)**
Pointer to an array of allocated descriptors.

**type (IN)**
Specifies the type of storage to be freed. See the types listed in `OCIDescriptorAlloc()`.

### Comments

An error is returned when a descriptor is allocated using `OCIDescriptorAlloc()`, but freed using `OCIArrayDescriptorFree()`.

If you perform LOB operations, you must always call `OCILobFreeTemporary()` before calling `OCIArrayDescriptorFree()` to free the contents of the temporary LOB. See About Freeing Temporary LOBs for more information.

Descriptors allocated using `OCIArrayDescriptorAlloc()` must be freed using `OCIArrayDescriptorFree()`. You must be careful to free the entire array at once: pass in the pointer `descpp` returned by `OCIArrayDescriptorAlloc()` to `OCIArrayDescriptorFree()` appropriately. Otherwise, there can be memory leaks.

### Related Topics

- OCIArrayDescriptorAlloc()
  Allocates an array of descriptors.
- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.

## 25.3.3 OCIAttrGet()

Gets the value of an attribute of a handle.

### Purpose

Gets the value of an attribute of a handle.

### Syntax

```
sword OCIAttrGet ( const void     *trgthndlp,
                   ub4             trghndltyp,
                   void           *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 4-1 or any descriptor in Table 4-2.

**attributep (OUT)**
Pointer to the storage for an attribute value. Is 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 `void` 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 handle types and their readable attributes are listed in Handle and Descriptor Attributes.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 that you are trying to describe. If the top-level parameter descriptor has an attribute that 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() and About Describing Select-List Items

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 that can be obtained through an `OCIAttrGet()` call.

Then an `OCIParamGet()` call 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 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 Topics**

- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIDescribeAny()
  Describes existing schema and subschema objects.
- OCIParamGet()
  Returns a descriptor of a parameter specified by position in the describe handle or statement handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.3.4 OCIAttrSet()

Sets the value of an attribute of a handle or a descriptor.

### Purpose

Sets the value of an attribute of a handle or a descriptor.

### Syntax

```
sword OCIAttrSet ( void        *trgthndlp,
                   ub4          trghndltyp,
                   void        *attributep,
                   ub4          size,
                   ub4          attrtype,
                   OCIError    *errhp );
```

### Parameters

**trgthndlp (IN/OUT)**
Pointer to a handle 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**Comments**

See Handle and Descriptor Attributes for a list of handle types and their writable attributes.

**Related Topics**

- OCIArrayDescriptorAlloc()
  Allocates an array of descriptors.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.3.5 OCIDescriptorAlloc()

Allocates storage to hold descriptors or LOB locators.

### Purpose

Allocates storage to hold descriptors or LOB locators.

### Syntax

```
sword OCIDescriptorAlloc ( const void    *parenth,
                           void         **descpp,
                           ub4            type,
                           size_t         xtramem_sz,
                           void         **usrmempp);
```

### Parameters

**parenth (IN)**
An environment handle.

**descpp (OUT)**
Returns a descriptor or LOB locator of the 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_PARAM` - Specifies generation of a read-only parameter descriptor of C type `OCIParam`.
- `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`.
- `OCI_DTYPE_AQNFY` - Specifies generation of an Advanced Queuing notification descriptor of C type `OCIAQNotify`.
- `OCI_DTYPE_AQLIS_OPTIONS` - Specifies generation of an Advanced Queuing listen descriptor of C type `OCIAQListenOpts`.
- `OCI_DTYPE_AQLIS_MSG_PROPERTIES` - Specifies generation of an Advanced Queuing message properties descriptor of C type `OCIAQLisMsgProps`.
- `OCI_DTYPE_SHARDING_KEY` - Specifies generation of the shard key or the shard group key of C type `OCIShardingKey`.
- `OCI_DTYPE_SRVDN` - Specifies generation of a Distinguished Names descriptor of C type `OCIServerDNs`.

- `OCI_DTYPE_UCB` - Specifies generation of a user callback descriptor of C type `OCIUcb`.

**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.

### Related Topics

- OCIDescriptorFree()
  Deallocates a previously allocated descriptor.
- OCIArrayDescriptorAlloc()
  Allocates an array of descriptors.
- OCIArrayDescriptorFree()
  Free a previously allocated array of descriptors.

> **See Also:**
>
> User Memory Allocation for more information about the `xtramem_sz` parameter and user memory allocation

## 25.3.6 OCIDescriptorFree()

Deallocates a previously allocated descriptor.

### Purpose

Deallocates a previously allocated descriptor.

### Syntax

```
sword OCIDescriptorFree ( void     *descp,
                          ub4       type );
```

### Parameters

**descp (IN)**
An allocated descriptor.

**type (IN)**
Specifies the type of storage to be freed. See the types listed in `OCIDescriptorAlloc()`.

### Comments

This call frees storage associated with a descriptor. Returns `OCI_SUCCESS` or `OCI_INVALID_HANDLE`. All descriptors can be explicitly deallocated; however, OCI deallocates a descriptor if the environment handle is deallocated.

If you perform LOB operations, you must always call `OCILobFreeTemporary()` before calling `OCIDescriptorFree()` to free the contents of the temporary LOB. See About Freeing Temporary LOBs for more information.

### Related Topics

- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.

## 25.3.7 OCIHandleAlloc()

Returns a pointer to an allocated and initialized handle.

### Purpose

Returns a pointer to an allocated and initialized handle.

### Syntax

```
sword OCIHandleAlloc ( const void    *parenth,
                       void         **hndlpp,
                       ub4            type,
                       size_t         xtramem_sz,
                       void         **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 Table 4-1.

**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 that 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.

**Related Topics**

- OCIHandleFree()
  Explicitly deallocates a handle

> **See Also:**
>
> User Memory Allocation for more information about using the `xtramem_sz` parameter for user memory allocation

## 25.3.8 OCIHandleFree()

Explicitly deallocates a handle

**Purpose**

This call explicitly deallocates a handle.

**Syntax**

```
sword OCIHandleFree ( void      *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 Table 4-1.

**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`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`.

All handles may be explicitly deallocated. The OCI deallocates a child handle if the parent is deallocated.

When a statement handle is freed, the cursor associated with the statement handle is closed, but the actual cursor closing may be deferred to the next round-trip to the server. If the application must close the cursor immediately, you can make a server round-trip call, such as `OCIServerVersion()` or `OCIPing()`, after the `OCIHandleFree()` call.

**Related Topics**

- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIServerVersion()
  Returns the Oracle Database version string.
- OCIPing()
  Confirms that the connection and the server are active.

## 25.3.9 OCIParamGet()

Returns a descriptor of a parameter specified by position in the describe handle or statement handle.

### Purpose

Returns a descriptor of a parameter specified by position in the describe handle or statement handle.

### Syntax

```
sword OCIParamGet ( const void        *hndlp,
                    ub4                htype,
                    OCIError          *errhp,
                    void             **parmdpp,
                    ub4                pos );
```

### Parameters

**hndlp (IN)**
A statement handle or describe handle. The `OCIParamGet()` function returns 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 is 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 leaks memory unless you explicitly free the parameter descriptor between each call to `OCIParamGet()`.

### Related Topics

- OCIArrayDescriptorAlloc()
  Allocates an array of descriptors.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIParamSet()
  Sets a complex object retrieval (COR) descriptor into a COR handle.
- OCIDescriptorFree()
  Deallocates a previously allocated descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> Handle and Descriptor Attributes for more detailed information about parameter descriptor attributes

## 25.3.10 OCIParamSet()

Sets a complex object retrieval (COR) descriptor into a COR handle.

### Purpose

Sets a complex object retrieval (COR) descriptor into a COR handle.

### Syntax

```
sword OCIParamSet ( void            *hndlp,
                    ub4              htype,
                    OCIError        *errhp,
                    const void      *dscp,
                     ub4             dtyp,
                     ub4             pos );
```

### Parameters

**hndlp (IN/OUT)**
Handle pointer.

**htype (IN)**
Handle type.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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()`.

### Related Topics

- OCIParamGet()
  Returns a descriptor of a parameter specified by position in the describe handle or statement handle.
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> Complex Object Retrieval for more information about complex object retrieval

# 25.4 Bind, Define, and Describe Functions

Lists and describes the bind, define, and describe functions.

Table 25-4 lists the bind, define, and describe functions that are described in this section.

**Table 25-4 Bind, Define, and Describe Functions**

| Function | Purpose |
|---|---|
| OCIBindArrayOfStruct() | Set skip parameters for static array bind |
| OCIBindByName() | Bind by name |
| OCIBindByName2() | Bind by name. Use when return lengths exceed `UB2MAXVAL` on the client. |
| OCIBindByPos() | Bind by position |
| OCIBindByPos2() | Bind by position. Use when return lengths exceed `UB2MAXVAL` on the client. |
| OCIBindDynamic() | Set additional attributes after bind with `OCI_DATA_AT_EXEC` mode |
| OCIBindObject() | Set additional attributes for bind of named data type |
| OCIDefineArrayOfStruct() | Set additional attributes for static array define |
| OCIDefineByPos() | Define an output variable association |
| OCIDefineByPos2() | Define an output variable association. Use when return lengths exceed `UB2MAXVAL` on the client. |
| OCIDefineDynamic() | Set additional attributes for define in `OCI_DYNAMIC_FETCH` mode |
| OCIDefineObject() | Set additional attributes for define of named data type |
| OCIDescribeAny() | Describe existing schema objects |
| OCIStmtGetBindInfo() | Get bind and indicator variable names and handle |

## 25.4.1 OCIBindArrayOfStruct()

Sets up the skip parameters for a static array bind.

### Purpose

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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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.

### Related Topics

- OCIBindByName()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIBindByName2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this function when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIBindByPos()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIBindByPos2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.

> **See Also:**
>
> About Binding and Defining Arrays of Structures in OCI for information about skip parameters

## 25.4.2 OCIBindByName()

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.

### 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 OraText   *placeholder,
                      sb4              placeh_len,
                      void            *valuep,
                      sb4              value_sz,
                      ub2              dty,
                      void            *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 that 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**placeholder (IN)**
The placeholder, specified by its name, that 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, the placeholder 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 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 `OCIStmtPrepare2()` , the default encoding for the string type `valuep` is in the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`, unless you call `OCIAttrSet()` to manually reset the character set for the bind handle.

> **See Also:**
>
> Bind Handle Attributes

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()`.

When `mode` is set to `OCI_IOV`, pass the base address of the `OCIIOV` struct.

**value_sz (IN)**
The maximum size possible in bytes of any data value (passed using `valuep`) for this bind variable. This size is always expected to be the size in bytes. 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 pointer to the specific type; for example, `sizeof (OCILobLocator *)`. The same applies even when mode is `OCI_IOV`.

**dty (IN)**
The data type of the values being bound. Named data types (`SQLT_NTY`) and `REF`s (`SQLT_REF`) are valid only if the application has been initialized in object mode. For named data types or `REF`s, additional calls must be made with the bind handle to set up the data type-specific attributes. See Comments for information about records, collections, and Booleans. For named data type `SQLT_CHR`, `OCIBindByName()` trims trailing blanks when the actual length is 0. Specify the actual length to prevent trailing blanks from being trimmed.

**indp (IN/OUT)**
Pointer to an indicator variable or array. For all data types except `SQLT_NTY`, this is a pointer to `sb2` or an array of `sb2`.
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

**alenp (IN/OUT)**
Pointer to the array of actual lengths of array elements.
When `OCIEnvNlsCreate()` (which is the recommended OCI environment handle creation interface) is used, then `alenp` lengths are consistently expected in bytes (for IN binds) and reported in bytes for OUT binds. The same treatment consistently also holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types. There are no special exceptions for UCS2 or for NCHAR cases.
When the older OCI environment handle creation interfaces are used (either `OCIEnvCreate()`) or deprecated `OCIEnvInit()`), `alenp` lengths are in bytes in general. However, `alenp` lengths are expected in characters for IN binds and also reported in characters for OUT binds only when either the character set is OCI_UC2ID (= OCI_UTF16ID) or when `OCI_ATTR_CHAR_COUNT` attribute is set on the corresponding OCIBind handle. The same treatment holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types.
This parameter is ignored for dynamic binds.

**rcodep (OUT)**
Pointer to the array of column-level return codes. This parameter is ignored for dynamic binds.

**maxarr_len (IN)**
A maximum array length parameter (the maximum possible number of elements the user's array can accommodate). Used only for PL/SQL indexed table bindings.

**curelep (IN/OUT)**
Current array length parameter (a pointer to the actual number of elements in the array before or after the execute operation). Used only for PL/SQL indexed table bindings.

**mode (IN)**
To maintain coding consistency, theoretically this parameter can take all three possible values used by `OCIStmtPrepare2()`. Because the encoding of bind variables should always be same as that of the statement containing this variable, an error is raised if you specify an encoding other than that of the statement. So the recommended setting for `mode` is `OCI_DEFAULT`, which makes the bind variable have the same encoding as its statement.
The valid modes are:

- `OCI_DEFAULT` - The default mode. The statement handle that `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 be provided at run time. 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 that 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:**
>
> Runtime Data Allocation and Piecewise Operations in OCI for more information about using the `OCI_DATA_AT_EXEC` mode

  When `mode` is set to `OCI_DATA_AT_EXEC`, do not provide values for `valuep`, `indp`, `alenp`, and `rcodep` in the main call. Pass zeros (0) for `indp` and `alenp`. Provide the values through the callback function registered using `OCIBindDynamic()`.

- `OCI_IOV` - Bind noncontiguous addresses of data. The `valuep` parameter must be of the type `OCIIOV *`. This mode is intended to be used for scatter or gather binding, which allows multiple buffers to be bound or defined to a position, for example column A for the first 10 rows in one buffer, next 5 rows in one buffer, and the remaining 25 rows in another buffer. That eliminates the need to allocate and copy all of them into one big buffer while doing the array execute operation.

> **See Also:**
>
> About Binding and Defining Multiple Buffers

When the allocated buffers are not required anymore, 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 that is being bound, and may also indicate the method by which data is provided at run time.

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.

The `OCIBindByName()` and `OCIBimdByName2()` 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 operation. Binding is *dynamic* when the IN bind data and the OUT bind buffers are provided by the application on demand at execution time to the client library. Dynamic binding is indicated by setting the `mode` parameter of this call to `OCI_DATA_AT_EXEC`.

> **See Also:**
>
> Runtime Data Allocation and Piecewise Operations in OCI for more information about dynamic binding

Both `OCIBindByName()` and `OCIBindByName2()` and `OCIBindByPos()` and `OCIBindByPos2()` take as a parameter a bind handle, which is implicitly allocated by the bind call. A separate bind handle is allocated for each placeholder the application is binding.

Additional bind calls may be required to specify particular attributes necessary when binding certain data types or handling input data in certain ways:

- If arrays of structures are being used, `OCIBindArrayOfStruct()` must be called to set up the necessary skip parameters.
- If data is being provided dynamically at run time, and the application uses user-defined callback functions, `OCIBindDynamic()` must be called to register the callbacks.
- If lengths in `alenp` greater than 64 Kilobytes (KB) are required, use `OCIBindDynamic()`.
- If a named data type is being bound, `OCIBindObject()` must be called to specify additional necessary information.
- If a statement with the `RETURNING` clause is used, a call to `OCIBindDynamic()` must follow this call.

With IN binds, the values for each element of the array, the actual lengths of each element, and the actual array length must be set up before the call to `OCIStmtExecute()`.

With OUT binds, the values for each element of the array, the actual lengths of each element, and the actual array length are returned from the server after the `OCIStmtExecute()` call.

**For Records**
Clients must bind package record types using `SQLT_NTY` as the DTY of the bind. In the OCI client, objects and records are represented as Named Types (NTY) and must use the same `SQLT` code.

**For Collections**
Clients must bind all package collection types using `SQLT_NTY`. This is the DTY used to bind all schema level collection types.

**For Booleans**
Clients must bind Boolean types (`OCI_TYPECODE_BOOLEAN`) using `SQLT_BOL`.

**Related Topics**

- OCIBindDynamic()
  Registers user callbacks for dynamic data allocation.
- OCIBindObject()
  Sets up additional attributes that are required for a named data type (object) bind.

- OCIBindArrayOfStruct()
  Sets up the skip parameters for a static array bind.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIStmtPrepare2()
  Prepares a SQL or PL/SQL statement for execution.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvInit()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIBindByName2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this function when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIBindByPos2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIStmtExecute()
  Associates an application request with a server.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.4.3 OCIBindByName2()

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this function when working with data types when actual lengths exceed `UB2MAXVAL` on the client.

### Purpose

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call instead of `OCIBindByName()` when working with data types when actual lengths exceed `UB2MAXVAL` on the client.

### Syntax

```
sword OCIBindByName2 ( OCIStmt         *stmtp,
                       OCIBind        **bindpp,
                       OCIError        *errhp,
                       const OraText   *placeholder,
                       sb4              placeh_len,
                       void            *valuep,
                       sb8              value_sz,
                       ub2              dty,
                       void            *indp,
ub4               *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 that 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**placeholder (IN)**
The placeholder, specified by its name, that 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, the placeholder 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 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 `OCIStmtPrepare2()`, the default encoding for the string type `valuep` is in the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`, unless you call `OCIAttrSet()` to manually reset the character set for the bind handle.

**See Also:**

Bind Handle Attributes

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()`.

When `mode` is set to `OCI_IOV`, pass the base address of the `OCIIOV` struct.

**value_sz (IN)**
The maximum size possible in bytes of any data value (passed using `valuep`) for this bind variable. This size is always expected to be the size in bytes. 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.
If the value of `value_sz` > `SB4MAXVAL`, an `ORA-24452` error will be issued, meaning that values > `SB4MAXVAL` are not supported in Release 12.1 or later.
For descriptors, locators, or REFs, whose size is unknown to client applications, use the size of the pointer to the specific type; for example, `sizeof (OCILobLocator *)`. The same applies even when mode is `OCI_IOV`.

**dty (IN)**
The data type of the values being bound. Named data types (`SQLT_NTY`) and `REF`s (`SQLT_REF`) are valid only if the application has been initialized in object mode. For named data types or `REF`s, additional calls must be made with the bind handle to set up the data type-specific attributes. See Comments for information about records, collections, and Booleans. For named data type `SQLT_CHR`, `OCIBindByName2()` trims trailing blanks when the actual length is 0. Specify the actual length to prevent trailing blanks from being trimmed.

**indp (IN/OUT)**
Pointer to an indicator variable or array. For all data types except `SQLT_NTY`, this is a pointer to `sb2` or an array of `sb2`.
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

**alenp (IN/OUT)**
Pointer to the array of actual lengths of array elements.
When `OCIEnvNlsCreate()` (which is the recommended OCI environment handle creation interface) is used, then `alenp` lengths are consistently expected in bytes (for IN binds) and reported in bytes for OUT binds. The same treatment consistently also holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types. There are no special exceptions for UCS2 or for NCHAR cases.
When the older OCI environment handle creation interfaces are used (either `OCIEnvCreate()`) or deprecated `OCIEnvInit()`), `alenp` lengths are in bytes in general. However, `alenp` lengths are expected in characters for IN binds and also reported in characters for OUT binds only when either the character set is OCI_UC2ID (= OCI_UTF16ID) or when `OCI_ATTR_CHAR_COUNT` attribute is set on the corresponding OCIBind handle. The same treatment holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types.
This parameter is ignored for dynamic binds.

**rcodep (OUT)**
Pointer to the array of column-level return codes. This parameter is ignored for dynamic binds.

**maxarr_len (IN)**
A maximum array length parameter (the maximum possible number of elements the user's array can accommodate). Used only for PL/SQL indexed table bindings.

**curelep (IN/OUT)**
Current array length parameter (a pointer to the actual number of elements in the array before or after the execute operation). Used only for PL/SQL indexed table bindings.

**mode (IN)**
To maintain coding consistency, theoretically this parameter can take all three possible values used by `OCIStmtPrepare2()`. Because the encoding of bind variables should always be same as that of the statement containing this variable, an error is raised if you specify an encoding other than that of the statement. So the recommended setting for `mode` is `OCI_DEFAULT`, which makes the bind variable have the same encoding as its statement.
The valid modes are:

- `OCI_DEFAULT` - The default mode. The statement handle that `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 be provided at run time. 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 that 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:**
  >
  > Runtime Data Allocation and Piecewise Operations in OCI for more information about using the `OCI_DATA_AT_EXEC` mode

  When `mode` is set to `OCI_DATA_AT_EXEC`, do not provide values for `valuep`, `indp`, `alenp`, and `rcodep` in the main call. Pass zeros (0) for `indp` and `alenp`. Provide the values through the callback function registered using `OCIBindDynamic()`.
- `OCI_IOV` - Bind noncontiguous addresses of data. The `valuep` parameter must be of the type `OCIIOV *`. This mode is intended to be used for scatter or gather binding, which allows multiple buffers to be bound or defined to a position, for example column A for the first 10 rows in one buffer, next 5 rows in one buffer, and the remaining 25 rows in another buffer. That eliminates the need to allocate and copy all of them into one big buffer while doing the array execute operation.

> **See Also:**
>
> About Binding and Defining Multiple Buffers

When the allocated buffers are not required anymore, 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 that is being bound, and may also indicate the method by which data is provided at run time.

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.

The `OCIBindByName2()` 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 `OCIBindByName2()`.

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 operation. Binding is *dynamic* when the IN bind data and the OUT bind buffers are provided by the application on demand at execution time to the client library. Dynamic binding is indicated by setting the `mode` parameter of this call to `OCI_DATA_AT_EXEC`.

> **See Also:**
>
> Runtime Data Allocation and Piecewise Operations in OCI for more information about dynamic binding

Both `OCIBindByName2()` and `OCIBindByPos2()` take as a parameter a bind handle, which is implicitly allocated by the bind call. A separate bind handle is allocated for each placeholder the application is binding.

Additional bind calls may be required to specify particular attributes necessary when binding certain data types or handling input data in certain ways:

- If arrays of structures are being used, `OCIBindArrayOfStruct()` must be called to set up the necessary skip parameters.
- If data is being provided dynamically at run time, and the application uses user-defined callback functions, `OCIBindDynamic()` must be called to register the callbacks.
- If lengths in `alenp` greater than 64 Kilobytes (KB) are required, use `OCIBindDynamic()`.
- If a named data type is being bound, `OCIBindObject()` must be called to specify additional necessary information.

- If a statement with the `RETURNING` clause is used, a call to `OCIBindDynamic()` must follow this call.

With IN binds, the values for each element of the array, the actual lengths of each element, and the actual array length must be set up before the call to `OCIStmtExecute()`.

With OUT binds, the values for each element of the array, the actual lengths of each element, and the actual array length are returned from the server after the `OCIStmtExecute()` call.

**For Records**
Clients must bind package record types using `SQLT_NTY` as the DTY of the bind. In the OCI client, objects and records are represented as Named Types (NTY) and must use the same `SQLT` code.

**For Collections**
Clients must bind all package collection types using `SQLT_NTY`. This is the DTY used to bind all schema level collection types.

**For Booleans**
Clients must bind Boolean types (`OCI_TYPECODE_BOOLEAN`) using `SQLT_BOL`.

**Related Topics**

- OCIBindDynamic()
  Registers user callbacks for dynamic data allocation.
- OCIBindObject()
  Sets up additional attributes that are required for a named data type (object) bind.
- OCIBindArrayOfStruct()
  Sets up the skip parameters for a static array bind.
- OCIBindByName()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIStmtPrepare2()
  Prepares a SQL or PL/SQL statement for execution.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvInit()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIBindByPos2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.

- OCIStmtExecute()
  Associates an application request with a server.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.4.4 OCIBindByPos()

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.

### 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,
                     void          *valuep,
                     sb4            value_sz,
                     ub2            dty,
                     void          *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 that 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**position (IN)**
The placeholder attributes are specified by position if `OCIBindByPos()` is being called. Positions are 1-based.

**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.
When `mode` is set to `OCI_IOV`, pass the base address of the `OCIIOV` struct.

> **See Also:**
>
> Bind Handle Attributes

**value_sz (IN)**
The maximum size possible in bytes of any data value (passed using `valuep`) for this bind variable. This size is always expected to be the size in bytes. 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 pointer to the specific type; for example, `sizeof (OCILobLocator *)`. The same applies even when mode is `OCI_IOV`.

**dty (IN)**
The data type of the values being bound. Named data types (SQLT_NTY) and `REF`s (SQLT_REF) are valid only if the application has been initialized in object mode. For named data types or `REF`s, additional calls must be made with the bind handle to set up the attributes specific to the data type. See Comments for information about records, collections, and Booleans. For named data type `SQLT_CHR`, `OCIBindByPos()` trims trailing blanks when the actual length is 0. Specify the actual length to prevent trailing blanks from being trimmed.

**indp (IN/OUT)**
Pointer to an indicator variable or array. For all data types, this is a pointer to `sb2` or an array of `sb2` values. The only exception is SQLT_NTY, where this pointer is ignored and the actual pointer to the indicator structure or an array of indicator structures is initialized by `OCIBindObject()`. The `indp` parameter 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

**alenp (IN/OUT)**
Pointer to an array of actual lengths of array elements.
When `OCIEnvNlsCreate()` (which is the recommended OCI environment handle creation interface) is used, then `alenp` lengths are consistently expected in bytes (for IN binds) and reported in bytes for OUT binds. The same treatment consistently also holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types. There are no special exceptions for UCS2 or for NCHAR cases.

When the older OCI environment handle creation interfaces are used (either `OCIEnvCreate()` or deprecated `OCIEnvInit()`), `alenp` lengths are in bytes in general. However, `alenp` lengths are expected in characters for IN binds and also reported in characters for OUT binds only when either the character set is OCI_UC2ID (= OCI_UTF16ID) or when `OCI_ATTR_CHAR_COUNT` attribute is set on the corresponding OCIBind handle. The same treatment holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types.
This parameter is ignored for dynamic binds.

**rcodep (OUT)**
Pointer to an array of column-level return codes. This parameter is ignored for dynamic binds.

**maxarr_len (IN)**
A maximum array length parameter (the maximum possible number of elements that the user's array can accommodate). Used only for PL/SQL indexed table bindings.

**curelep (IN/OUT)**
Current array length parameter (a pointer to the actual number of elements in the array before or after the execute operation). Used only for PL/SQL indexed table bindings.

**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 be provided at run time. 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 following ways:
  - Callbacks using a user-defined function that must be registered with a subsequent call to `OCIBindDynamic()`.
  - A polling mechanism using calls supplied by OCI. This mode is assumed if no callbacks are defined.

  > **See Also:**
  >
  > Runtime Data Allocation and Piecewise Operations in OCI for more information about using the `OCI_DATA_AT_EXEC` mode

  When `mode` is set to `OCI_DATA_AT_EXEC`, do not provide values for `valuep`, `indp`, `alenp`, and `rcodep` in the main call. Pass zeros (0) for `indp` and `alenp`.

Provide the values through the callback function registered using `OCIBindDynamic()`.

- `OCI_IOV` - Bind noncontiguous addresses of data. The `valuep` parameter must be of the type `OCIIOV *`. This mode is intended to be used for scatter or gather binding, which allows multiple buffers to be bound or defined to a position, for example column A for the first 10 rows in one buffer, next 5 rows in one buffer, and the remaining 25 rows in another buffer. That eliminates the need to allocate and copy all of them into one big buffer while doing the array execute operation.

> **See Also:**
>
> About Binding and Defining Multiple Buffers

When the allocated buffers are not required anymore, 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 that is being bound, and may also indicate the method by which data is to be provided at run time.

This function also implicitly allocates the bind handle indicated by the `bindpp` parameter. If a non-`NULL` pointer is passed in `**bindpp`, 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 operation. Binding is *dynamic* when the IN bind data and the OUT bind buffers are provided by the application on demand at execution time to the client library. Dynamic binding is indicated by setting the `mode` parameter of this call to `OCI_DATA_AT_EXEC`.

> **See Also:**
>
> Runtime Data Allocation and Piecewise Operations in OCI for more information about dynamic binding

Both `OCIBindByName()` and `OCIBindByPos()` take as a parameter a bind handle, which is implicitly allocated by the bind call. A separate bind handle is allocated for each placeholder the application is binding.

Additional bind calls may be required to specify particular attributes necessary when binding certain data types or handling input data in certain ways:

- If arrays of structures are being used, `OCIBindArrayOfStruct()` must be called to set up the necessary skip parameters.

- If data is being provided dynamically at run time, and the application uses user-defined callback functions, `OCIBindDynamic()` must be called to register the callbacks.
- If lengths in `alenp` greater than 64 KB are required, use `OCIBindDynamic()`.
- If a named data type is being bound, `OCIBindObject()` must be called to specify additional necessary information.
- If a statement with the `RETURNING` clause is used, a call to `OCIBindDynamic()` must follow this call.

With IN binds, the values for each element of the array, the actual lengths of each element, and the actual array length must be set up before the call to `OCIStmtExecute()`.

With OUT binds, the values for each element of the array, the actual lengths of each element, and the actual array length are returned from the server after the`OCIStmtExecute()` call.

**For Records**
Clients must bind package record types using `SQLT_NTY` as the DTY of the bind. In the OCI client, objects and records are represented as Named Types (NTY) and must use the same `SQLT` code.

**For Collections**
Clients must bind all package collection types using `SQLT_NTY`. This is the DTY used to bind all schema level collection types.

**For Booleans**
Clients must bind Boolean types (`OCI_TYPECODE_BOOLEAN`) using `SQLT_BOL`.

**Related Topics**

- OCIBindDynamic()
  Registers user callbacks for dynamic data allocation.
- OCIBindObject()
  Sets up additional attributes that are required for a named data type (object) bind.
- OCIBindArrayOfStruct()
  Sets up the skip parameters for a static array bind.
- OCIBindByPos2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvInit()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.

- OCIBindByName()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIStmtExecute()
  Associates an application request with a server.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.4.5 OCIBindByPos2()

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.

### Purpose

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call instead of `OCIBindByPos()` when working with data types when actual lengths exceed `UB2MAXVAL` on the client.

### Syntax

```
sword OCIBindByPos2 ( OCIStmt      *stmtp,
                      OCIBind     **bindpp,
                      OCIError     *errhp,
                      ub4           position,
                      void         *valuep,
                      sb8           value_sz,
                      ub2           dty,
                      void         *indp,
                      ub4          *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 that 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**position (IN)**
The placeholder attributes are specified by position if `OCIBindByPos()` is being called. Positions are 1-based.

**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.
When `mode` is set to `OCI_IOV`, pass the base address of the `OCIIOV` struct.

**See Also:**

Bind Handle Attributes

**value_sz (IN)**
The maximum size possible in bytes of any data value (passed using `valuep`) for this bind variable. This size is always expected to be the size in bytes. 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.
If the value of `value_sz` > `SB4MAXVAL`, an `ORA-24452` error will be issued, meaning that values > `SB4MAXVAL` are not supported in Release 12.1 or later.
For descriptors, locators, or `REF`s, whose size is unknown to client applications, use the size of the pointer to the specific type; for example, `sizeof` (`OCILobLocator *`). The same applies even when mode is `OCI_IOV`.

**dty (IN)**
The data type of the values being bound. Named data types (SQLT_NTY) and `REF`s (SQLT_REF) are valid only if the application has been initialized in object mode. For named data types or `REF`s, additional calls must be made with the bind handle to set up the attributes specific to the data type. See Comments for information about records, collections, and Booleans. For named data type `SQLT_CHR`, `OCIBindByPos2()` trims trailing blanks when the actual length is 0. Specify the actual length to prevent trailing blanks from being trimmed.

**indp (IN/OUT)**
Pointer to an indicator variable or array. For all data types, this is a pointer to `sb2` or an array of `sb2` values. The only exception is SQLT_NTY, where this pointer is ignored and the actual pointer to the indicator structure or an array of indicator structures is initialized by `OCIBindObject()`. The `indp` parameter 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

**alenp (IN/OUT)**
Pointer to an array of actual lengths of array elements.
When `OCIEnvNlsCreate()` (which is the recommended OCI environment handle creation interface) is used, then `alenp` lengths are consistently expected in bytes (for IN binds) and reported in bytes for OUT binds. The same treatment consistently also holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types. There are no special exceptions for UCS2 or for NCHAR cases.
When the older OCI environment handle creation interfaces are used (either `OCIEnvCreate()` or deprecated `OCIEnvInit()`), `alenp` lengths are in bytes in general. However, `alenp` lengths are expected in characters for IN binds and also reported in characters for OUT binds only when either the character set is OCI_UC2ID (= OCI_UTF16ID) or when `OCI_ATTR_CHAR_COUNT` attribute is set on the corresponding OCIBind handle. The same treatment holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types.
This parameter is ignored for dynamic binds.

**rcodep (OUT)**
Pointer to an array of column-level return codes. This parameter is ignored for dynamic binds.

**maxarr_len (IN)**
A maximum array length parameter (the maximum possible number of elements that the user's array can accommodate). Used only for PL/SQL indexed table bindings.

**curelep (IN/OUT)**
Current array length parameter (a pointer to the actual number of elements in the array before or after the execute operation). Used only for PL/SQL indexed table bindings.

**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 be provided at run time. 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 following ways:
  - Callbacks using a user-defined function that must be registered with a subsequent call to `OCIBindDynamic()`.
  - A polling mechanism using calls supplied by OCI. This mode is assumed if no callbacks are defined.

> **See Also:**
>
> Runtime Data Allocation and Piecewise Operations in OCI for more information about using the `OCI_DATA_AT_EXEC` mode

When `mode` is set to `OCI_DATA_AT_EXEC`, do not provide values for `valuep`, `indp`, `alenp`, and `rcodep` in the main call. Pass zeros (0) for `indp` and `alenp`. Provide the values through the callback function registered using `OCIBindDynamic()`.

- `OCI_IOV` - Bind noncontiguous addresses of data. The `valuep` parameter must be of the type `OCIIOV *`. This mode is intended to be used for scatter or gather binding, which allows multiple buffers to be bound or defined to a position, for example column A for the first 10 rows in one buffer, next 5 rows in one buffer, and the remaining 25 rows in another buffer. That eliminates the need to allocate and copy all of them into one big buffer while doing the array execute operation.

  > **See Also:**
  >
  > About Binding and Defining Multiple Buffers

When the allocated buffers are not required anymore, 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 that is being bound, and may also indicate the method by which data is to be provided at run time.

This function also implicitly allocates the bind handle indicated by the `bindpp` parameter. If a non-`NULL` pointer is passed in `**bindpp`, OCI assumes that this points to a valid handle that has been previously allocated with a call to `OCIHandleAlloc()` or `OCIBindByPos2()`.

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 operation. Binding is *dynamic* when the IN bind data and the OUT bind buffers are provided by the application on demand at execution time to the client library. Dynamic binding is indicated by setting the `mode` parameter of this call to `OCI_DATA_AT_EXEC`.

> **See Also:**
>
> Runtime Data Allocation and Piecewise Operations in OCI for more information about dynamic binding

Both `OCIBindByName2()` and `OCIBindByPos2()` take as a parameter a bind handle, which is implicitly allocated by the bind call. A separate bind handle is allocated for each placeholder the application is binding.

Additional bind calls may be required to specify particular attributes necessary when binding certain data types or handling input data in certain ways:

- If arrays of structures are being used, `OCIBindArrayOfStruct()` must be called to set up the necessary skip parameters.
- If data is being provided dynamically at run time, and the application uses user-defined callback functions, `OCIBindDynamic()` must be called to register the callbacks.
- If lengths in `alenp` greater than 64 KB are required, use `OCIBindDynamic()`.
- If a named data type is being bound, `OCIBindObject()` must be called to specify additional necessary information.
- If a statement with the `RETURNING` clause is used, a call to `OCIBindDynamic()` must follow this call.

With IN binds, the values for each element of the array, the actual lengths of each element, and the actual array length must be set up before the call to `OCIStmtExecute()` .

With OUT binds, the values for each element of the array, the actual lengths of each element, and the actual array length are returned from the server after the `OCIStmtExecute()` call.

**For Records**
Clients must bind package record types using `SQLT_NTY` as the DTY of the bind. In the OCI client, objects and records are represented as Named Types (NTY) and must use the same `SQLT` code.

**For Collections**
Clients must bind all package collection types using `SQLT_NTY`. This is the DTY used to bind all schema level collection types.

**For Booleans**
Clients must bind Boolean types (`OCI_TYPECODE_BOOLEAN`) using `SQLT_BOL`.

**Related Topics**

- OCIBindDynamic()
  Registers user callbacks for dynamic data allocation.
- OCIBindObject()
  Sets up additional attributes that are required for a named data type (object) bind.
- OCIBindArrayOfStruct()
  Sets up the skip parameters for a static array bind.
- OCIBindByPos()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIEnvInit()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIBindByName2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this function when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIStmtExecute()
  Associates an application request with a server.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.4.6 OCIBindDynamic()

Registers user callbacks for dynamic data allocation.

### Purpose

Registers user callbacks for dynamic data allocation.

### Syntax

```
sword OCIBindDynamic ( OCIBind     *bindp,
                       OCIError    *errhp,
                       void        *ictxp,
                       OCICallbackInBind         (icbfp)(
                                void             *ictxp,
                                OCIBind          *bindp,
                                ub4               iter,
                                ub4               index,
                                void            **bufpp,
                                ub4              *alenp,
                                ub1              *piecep,
                                void            **indpp ),
                                void             *octxp,
                       OCICallbackOutBind        (ocbfp)(
                                void             *octxp,
                                OCIBind          *bindp,
                                ub4               iter,
                                ub4               index,
                                void            **bufpp,
                                ub4             **alenpp,
                                ub1              *piecep,
                                void            **indpp,
                                ub2             **rcodepp ) );
```

### Parameters

**bindp (IN/OUT)**
A bind handle returned by a call to `OCIBindByName()` or `OCIBindByPos()`.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**ictxp (IN)**
The context pointer required by the callback function `icbfp`.

**icbfp (IN)**
The callback function that 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)**
A 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 the `curelep` parameter of the bind call.

**bufpp (OUT)**
The pointer to the buffer or storage. For descriptors, `*bufpp` contains a pointer to the descriptor. For example, if you define the following parameter, then you set `*bufpp` to `lobp`, not `*lobp`.

```
OCILobLocator    *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

**alenp (OUT)**
A pointer to storage for OCI to fill in the size of the bind value or piece after it has been read. For descriptors, pass the size of the pointer to the descriptor; for example, `sizeof(OCILobLocator *)`.

**piecep (OUT)**
A 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 data types that do not support piecewise operations, you must pass `OCI_ONE_PIECE` or an error is generated.

**indpp (OUT)**
Contains the indicator value. This is either a pointer to an sb2 value or a pointer to an indicator structure for binding named data types.

**octxp (IN)**
The context pointer required by the callback function `ocbfp()`.

**ocbfp (IN)**
The callback function that 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 callback function.

**bindp (IN)**
The bind handle passed in to uniquely identify this bind variable.

**iter (IN)**
A 0-based execute iteration value.

**index (IN)**
For PL/SQL, the 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 the `curelep` parameter of the bind call.

**bufpp (OUT)**
A pointer to a buffer to write the bind value or piece in.
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, see "Bind Handle Attributes".

**alenpp (IN/OUT)**
A pointer to storage for OCI to fill in the size of the bind value or 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 code points.

**piecep (IN/OUT)**
Returns a piece value from the callback (application) to the Oracle Database, 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)**
Contains the indicator value. This is either a pointer to an `sb2` value, or a pointer to an indicator structure for binding named data types.

**rcodepp (OUT)**
Returns a pointer to 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 the call is successful. Any return code other than `OCI_CONTINUE` signals that the client wants to terminate processing immediately.

> **See Also:**
>
> Runtime Data Allocation and Piecewise Operations in OCI for more information about the `OCI_DATA_AT_EXEC` mode

When passing the address of a storage area, ensure that the storage area exists even after the application returns from the callback. This means that you should not allocate such storage on the stack.

> **Note:**
>
> After you use `OCIEnvNlsCreate()` to create the environment handle, the actual lengths and returned lengths of bind and define handles are always in number of bytes.

**Related Topics**

- OCIBindByName()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIBindByName2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this function when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIBindByPos()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIBindByPos2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.4.7 OCIBindObject()

Sets up additional attributes that are required for a named data type (object) bind.

**Purpose**

Sets up additional attributes that are required for a named data type (object) bind.

### Syntax

```
sword OCIBindObject ( OCIBind          *bindp,
                      OCIError         *errhp,
                      const OCIType    *type,
                      void            **pgvpp,
                      ub4              *pvszsp,
                      void            **indpp,
                      ub4              *indszp, );
```

### Parameters

**bindp (IN/OUT)**
The bind handle returned by the call to `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**type (IN)**
Points to the TDO that describes the type of 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 data type value or `REF` is allocated in the Object Cache, and a `REF` is returned.
`pgvpp` is ignored if the `OCI_DATA_AT_EXEC` mode is set. Then the named data type buffers are requested at run time. For static array binds, skip factors may be specified using the `OCIBindArrayOfStruct()` call. The skip factors are used to compute the address of the next pointer to the value, the indicator structure, and their sizes.

**pvszsp (OUT) [optional]**
Points to the size of the program variable. The size of the named data type is not required on input. For an array, `pvszsp` is an array of `ub4`s. On return, for OUT bind variables, this points to sizes of the named data types 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 run time.

**indpp (IN/OUT) [optional]**
Address of the program variable buffer containing the parallel indicator structure. For an array, `indpp` 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 operation when all OUT values have been received, `indpp` points to the pointers of these newly allocated indicator structures. Required only for SQLT_NTY binds. The `indpp` parameter is ignored if the `OCI_DATA_AT_EXEC` mode is set. Then the indicator is requested at run time.

**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 run time.

**Comments**

This function sets up additional attributes for binding a named data type or a `REF`. An error is 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 data type `OCIType` for the named data type being defined. The TDO can be retrieved with a call to `OCITypeByName()`.

If the `OCI_DATA_AT_EXEC` mode was specified in `OCIBindByName()` or `OCIBindByName2()` or `OCIBindByPos()` or `OCIBindByPos2()`, 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()`
- 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 Topics**

- OCIBindByName()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIBindByName2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this function when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIBindByPos()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIBindByPos2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIBindArrayOfStruct()
  Sets up the skip parameters for a static array bind.
- OCITypeByName()
  Gets the most current version of an existing TDO.
- OCIStmtSetPieceInfo()
  Sets piece information for a piecewise operation.
- OCIStmtExecute()
  Associates an application request with a server.
- OCIObjectFree()
  Frees and unpins an object instance.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.4.8 OCIDefineArrayOfStruct()

Specifies additional attributes necessary for a static array define, used in an array of structures (multirow, multicolumn) fetch.

### Purpose

Specifies additional attributes necessary for a static array define, used in an array of structures (multirow, multicolumn) 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 that was returned by a call to `OCIDefineByPos()` or `OCIDefineByPos2()`.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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()` or `OCIDefineByPos2()`. If the application is binding an array of structures involving objects, it must call `OCIDefineObject()` first, and then call `OCIDefineArrayOfStruct()`.

### Related Topics

- OCIDefineByPos()
  Associates an item in a select list with the type and output data buffer.
- OCIDefineByPos2()
  Associates an item in a select list with the type and output data buffer. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.

- OCIDefineObject()
  Sets up additional attributes necessary for a named data type or `REF` define.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> Skip Parameters

## 25.4.9 OCIDefineByPos()

Associates an item in a select list with the type and output data buffer.

### 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,
                       void        *valuep,
                       sb4          value_sz,
                       ub2          dty,
                       void        *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. For 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:**
>
> You 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 will be returned.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**position (IN)**
The position of this value in the select list. Positions are 1-based and are numbered from left to right. The value 0 selects `ROWIDs` (the globally unique identifier for a row in a table).

**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.
When `mode` is set to `OCI_IOV`, pass the base address of the `OCIIOV` struct.

**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, can be specified by using `OCIAttrSet()`.
In a multibyte conversion environment, a truncation error is generated if the number of bytes specified is insufficient to handle the number of characters needed.
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.
When `mode` is set to `OCI_IOV`, pass the size of the data value.

> **See Also:**
>
> Bind Handle Attributes

**dty (IN)**
The data type. Named data type (SQLT_NTY) and `REF` (SQLT_REF) are valid only if the environment has been initialized in object mode.
SQLT_CHR and SQLT_LNG can be specified for `CLOB` columns, and SQLT_BIN and SQLT_LBI can be specified for `BLOB` columns.

> **See Also:**
>
> Data Types for a listing of data type codes and values

**indp (IN)**
Pointer to an indicator variable or array. For scalar data types, pointer to `sb2` or an array of `sb2`s. Ignored for SQLT_NTY defines. For SQLT_NTY defines, a pointer to a named data type indicator structure or an array of named data type indicator structures is associated by a subsequent `OCIDefineObject()` call.

> **See Also:**
>
> Indicator Variables

**rlenp (IN/OUT)**
Pointer to array of length of data fetched.
When `OCIEnvNlsCreate()` (which is the recommended OCI environment handle creation interface) is used, then `rlenp` lengths are consistently reported in bytes. The same treatment consistently also holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types. There are no special exceptions for UCS2 or for NCHAR cases.
When the older OCI environment handle creation interfaces are used (either `OCIEnvCreate()` or deprecated `OCIEnvInit()`), `rlenp` lengths are in bytes in general. However, `rlenp` lengths are reported in characters when either the character set is OCI_UC2ID (= OCI_UTF16ID) or when `OCI_ATTR_CHAR_COUNT` attribute is set on the corresponding OCIBind handle. The same treatment holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types.

**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 such as `dty` or `value_sz` is changed from the previous define, this mode is ignored. Unexpected behavior results if an invalid 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 is to be provided at run time. When the client library needs a buffer to return the fetched data, the callback is invoked to provide a runtime buffer into which a piece or all the data is returned.

  > **See Also:**
  >
  > Implicit Fetching of ROWIDs

- `OCI_IOV` - Define noncontiguous addresses of data. The `valuep` parameter must be of the type `OCIIOV *`. This mode is intended to be used for scatter or gather binding, which allows multiple buffers to be bound or defined to a position, for example column A for the first 10 rows in one buffer, next 5 rows in one buffer, and the remaining 25 rows in another buffer. That eliminates the need to allocate and copy all of them into one big buffer while doing the array execute operation.

  > **See Also:**
  >
  > About Binding and Defining Multiple Buffers

**Comments**

This call defines an output buffer that receives data retrieved from Oracle Database. The define is a local step that is necessary when a `SELECT` statement returns data to your OCI application.

This call also implicitly allocates the define handle for the select-list item. If a non-`NULL` pointer is passed in `*defnpp`, OCI assumes that this points to a valid handle that has been previously allocated with a call to `OCIHandleAlloc()` or `OCIDefineByPos()` or `OCIDefineByPos2()`. This would be true for an application that is redefining a handle to a different address so that 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()` or `OCIDefineByPos2()`, which defines the minimal attributes required to specify the fetch.

Following the call to `OCIDefineByPos()` or `OCIDefineByPos2()` additional define calls may be necessary for certain data types or fetch modes:

- A call to `OCIDefineArrayOfStruct()` is necessary to set up skip parameters for an array fetch of multiple columns.
- A call to `OCIDefineObject()` is necessary to set up the appropriate attributes of a named data type (that is, object or collection) or `REF` fetch. In this case, the data buffer pointer in `OCIDefineByPos()` or `OCIDefineByPos2()` is ignored.
- Both `OCIDefineArrayOfStruct()` and `OCIDefineObject()` must be called after `OCIDefineByPos()` or `OCIDefineByPos2()` to fetch multiple rows with a column of named data types.

For a LOB define, the buffer pointer must be a pointer to a LOB locator of type `OCILobLocator`, allocated by the `OCIDescriptorAlloc()` call. LOB locators, and not LOB values, are always returned for a LOB column. LOB values can then be fetched using OCI LOB calls on the fetched locator. This same mechanism applies for all descriptor data types.

For NCHAR (fixed and varying length), the buffer pointer must point to an array of bytes sufficient for holding the required NCHAR characters.

Nested table columns are defined and fetched like any other named data type.

When defining an array of descriptors or locators, you should pass in an array of pointers to descriptors or locators.

When doing an array define for character columns, you should pass in an array of character buffers.

If the `mode` parameter in this call is set to `OCI_DYNAMIC_FETCH`, the client application can fetch data dynamically at run time. Runtime data can be provided in one of two ways:

- Callbacks using a user-defined function that must be registered with a subsequent call to `OCIDefineDynamic()`. When the client library needs a buffer to return the fetched data, the callback is invoked and the runtime buffers provided return a piece or all of the data.
- A polling mechanism using calls supplied by 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:**

- Runtime Data Allocation and Piecewise Operations in OCI for more information about using the `OCI_DYNAMIC_FETCH` mode
- Overview of Defining in OCI for more information about defines
- Implicit Fetching of ROWIDs

**Related Topics**

- OCIDefineArrayOfStruct()
  Specifies additional attributes necessary for a static array define, used in an array of structures (multirow, multicolumn) fetch.
- OCIDefineDynamic()
  Sets the additional attributes required if the `OCI_DYNAMIC_FETCH` mode was selected in `OCIDefineByPos()` or `OCIDefineByPos2()`.
- OCIDefineObject()
  Sets up additional attributes necessary for a named data type or `REF` define.
- OCIDefineByPos2()
  Associates an item in a select list with the type and output data buffer. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvInit()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.4.10 OCIDefineByPos2()

Associates an item in a select list with the type and output data buffer. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.

**Purpose**

Associates an item in a select list with the type and output data buffer. Use this call instead of `OCIDefineByPos()` when working with data types when actual lengths exceed `UB2MAXVAL` on the client.

### Syntax

```
sword OCIDefineByPos2 ( OCIStmt     *stmtp,
                        OCIDefine  **defnpp,
                        OCIError    *errhp,
                        ub4          position,
                        void        *valuep,
                        sb8          value_sz,
                        ub2          dty,
                        void        *indp,
                        ub4         *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. For 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:**
>
> You 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 will be returned.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**position (IN)**
The position of this value in the select list. Positions are 1-based and are numbered from left to right. The value 0 selects `ROWID`s (the globally unique identifier for a row in a table).

**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.
When `mode` is set to `OCI_IOV`, pass the base address of the `OCIIOV` struct.

**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, can be specified as additional bytes by using `OCIAttrSet()`.

If the value of `value_sz` > `SB4MAXVAL`, an `ORA-24452` error will be issued, meaning that values > `SB4MAXVAL` are not supported in Release 12.1 or later.

In a multibyte conversion environment, a truncation error is generated if the number of bytes specified is insufficient to handle the number of characters needed.

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.

When `mode` is set to `OCI_IOV`, pass the size of the data value.

> **See Also:**
>
> Bind Handle Attributes

**dty (IN)**

The data type. Named data type (SQLT_NTY) and `REF` (SQLT_REF) are valid only if the environment has been initialized in object mode.

SQLT_CHR and SQLT_LNG can be specified for `CLOB` columns, and SQLT_BIN and SQLT_LBI can be specified for `BLOB` columns.

> **See Also:**
>
> Data Types for a listing of data type codes and values

**indp (IN)**

Pointer to an indicator variable or array. For scalar data types, pointer to `sb2` or an array of `sb2`s. Ignored for SQLT_NTY defines. For SQLT_NTY defines, a pointer to a named data type indicator structure or an array of named data type indicator structures is associated by a subsequent `OCIDefineObject()` call.

> **See Also:**
>
> Indicator Variables

**rlenp (IN/OUT)**

Pointer to array of length of data fetched.

When `OCIEnvNlsCreate()` (which is the recommended OCI environment handle creation interface) is used, then `rlenp` lengths are consistently reported in bytes. The same treatment consistently also holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types. There are no special exceptions for UCS2 or for NCHAR cases.

When the older OCI environment handle creation interfaces are used (either `OCIEnvCreate()` or deprecated `OCIEnvInit()`), `rlenp` lengths are in bytes in general. However, `rlenp` lengths are reported in characters when either the character set is OCI_UC2ID (= OCI_UTF16ID) or when `OCI_ATTR_CHAR_COUNT` attribute is set on the corresponding OCIBind handle. The same treatment holds for the length prefix in `SQLT_VCS` (2-byte length prefix) and `SQLT_LVC` (4-byte length prefix) types.

**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 such as `dty` or `value_sz` is changed from the previous define, this mode is ignored. Unexpected behavior results if an invalid 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 is to be provided at run time. When the client library needs a buffer to return the fetched data, the callback is invoked to provide a runtime buffer into which a piece or all the data is returned.

  > **See Also:**
  >
  > Implicit Fetching of ROWIDs

- `OCI_IOV` - Define noncontiguous addresses of data. The `valuep` parameter must be of the type `OCIIOV *`. This mode is intended to be used for scatter or gather binding, which allows multiple buffers to be bound or defined to a position, for example column A for the first 10 rows in one buffer, next 5 rows in one buffer, and the remaining 25 rows in another buffer. That eliminates the need to allocate and copy all of them into one big buffer while doing the array execute operation.

  > **See Also:**
  >
  > About Binding and Defining Multiple Buffers

### Comments

This call defines an output buffer that receives data retrieved from Oracle Database. The define is a local step that is necessary when a `SELECT` statement returns data to your OCI application.

This call also implicitly allocates the define handle for the select-list item. If a non-`NULL` pointer is passed in `*defnpp`, OCI assumes that this points to a valid handle that has been previously allocated with a call to `OCIHandleAlloc()` or `OCIDefineByPos2()`. This would be true for an application that is redefining a handle to a different address so that 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 `OCIDefineByPos2()`, which defines the minimal attributes required to specify the fetch.

Following the call to `OCIDefineByPos2()` additional define calls may be necessary for certain data types or fetch modes:

- A call to `OCIDefineArrayOfStruct()` is necessary to set up skip parameters for an array fetch of multiple columns.
- A call to `OCIDefineObject()` is necessary to set up the appropriate attributes of a named data type (that is, object or collection) or `REF` fetch. In this case, the data buffer pointer in `OCIDefineByPos2()` is ignored.
- Both `OCIDefineArrayOfStruct()` and `OCIDefineObject()` must be called after `OCIDefineByPos2()` to fetch multiple rows with a column of named data types.

For a LOB define, the buffer pointer must be a pointer to a LOB locator of type `OCILobLocator`, allocated by the `OCIDescriptorAlloc()` call. LOB locators, and not LOB values, are always returned for a LOB column. LOB values can then be fetched using OCI LOB calls on the fetched locator. This same mechanism applies for all descriptor data types.

For NCHAR (fixed and varying length), the buffer pointer must point to an array of bytes sufficient for holding the required NCHAR characters.

Nested table columns are defined and fetched like any other named data type.

When defining an array of descriptors or locators, you should pass in an array of pointers to descriptors or locators.

When doing an array define for character columns, you should pass in an array of character buffers.

If the `mode` parameter in this call is set to `OCI_DYNAMIC_FETCH`, the client application can fetch data dynamically at run time. Runtime data can be provided in one of two ways:

- Callbacks using a user-defined function that must be registered with a subsequent call to `OCIDefineDynamic()`. When the client library needs a buffer to return the fetched data, the callback is invoked and the runtime buffers provided return a piece or all of the data.
- A polling mechanism using calls supplied by 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:**
>
> - Runtime Data Allocation and Piecewise Operations in OCI for more information about using the `OCI_DYNAMIC_FETCH` mode
> - Overview of Defining in OCI for more information about defines
> - Implicit Fetching of ROWIDs

**Related Topics**

- OCIDefineArrayOfStruct()
  Specifies additional attributes necessary for a static array define, used in an array of structures (multirow, multicolumn) fetch.

- OCIDefineDynamic()
  Sets the additional attributes required if the `OCI_DYNAMIC_FETCH` mode was selected in `OCIDefineByPos()` or `OCIDefineByPos2()`.
- OCIDefineObject()
  Sets up additional attributes necessary for a named data type or `REF` define.
- OCIDefineByPos()
  Associates an item in a select list with the type and output data buffer.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvInit()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 25.4.11 OCIDefineDynamic()

Sets the additional attributes required if the `OCI_DYNAMIC_FETCH` mode was selected in `OCIDefineByPos()` or `OCIDefineByPos2()`.

**Purpose**

Sets the additional attributes required if the `OCI_DYNAMIC_FETCH` mode was selected in `OCIDefineByPos()` or `OCIDefineByPos2()`.

**Syntax**

```
sword OCIDefineDynamic ( OCIDefine   *defnp,
                         OCIError    *errhp,
                         void        *octxp,
                         OCICallbackDefine      (ocbfp)(
                                  void          *octxp,
                                  OCIDefine     *defnp,
                                  ub4            iter,
                                  void         **bufpp,
                                  ub4          **alenpp,
                                  ub1           *piecep,
                                  void         **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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**octxp (IN)**
Points to a context for the callback function.

**ocbfp (IN)**
Points to a callback function. This is invoked at run time to get a pointer to the buffer into which the fetched data or a piece of it is to be retrieved. The callback also specifies the indicator, the return code, and the lengths of the data piece and indicator.

> **Note:**
>
> Normally, in an OCI function, an `IN` parameter refers to data being passed to OCI, and an `OUT` parameter refers to data coming back from OCI. For callbacks, this is reversed. `IN` means that data is coming from OCI into the callback, and `OUT` means that data is coming out of the callback and going to OCI.

The callback parameters are:

**octxp (IN/OUT)**
A context pointer passed as an argument to all the callback functions. When the client library needs a buffer to return the fetched data, the callback is invoked and the runtime buffers provided return a piece or all of the data.

**defnp (IN)**
The define handle.

**iter (IN)**
Specifies 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. If the buffer length provided in the first call is insufficient to store all the data returned by the server, then the callback is called again, and so on.

**piecep (IN/OUT)**
Returns a piece value from the callback (application) to OCI, as follows:

The `piecep` parameter indicates whether the piece to be fetched is the first piece, `OCI_FIRST_PIECE`, a subsequent piece, `OCI_NEXT_PIECE`, or the last piece, `OCI_LAST_PIECE`. The program can process the piece the next time the callback is called, or after the series of callbacks is over.

- `IN` - The value can be `OCI_ONE_PIECE`, `OCI_FIRST_PIECE`, or `OCI_NEXT_PIECE`.
- `OUT` - Depends on the IN value:
  - The `OUT` value can be `OCI_ONE_PIECE` if the `IN` value was `OCI_ONE_PIECE`.
  - The `OUT` value can be `OCI_ONE_PIECE` or `OCI_FIRST_PIECE` if the `IN` value was `OCI_FIRST_PIECE`.
  - The `OUT` value can be `OCI_NEXT_PIECE` or `OCI_LAST_PIECE` if the `IN` value was `OCI_NEXT_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()` or `OCIDefineByPos2()`. 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()`).

> **Note:**
>
> After you use `OCIEnvNlsCreate()` to create the environment handle, the actual lengths and returned lengths of bind and define handles are always in number of bytes.

### Related Topics

- OCIDefineObject()
  Sets up additional attributes necessary for a named data type or `REF` define.
- OCIBindDynamic()
  Registers user callbacks for dynamic data allocation.
- OCIDefineByPos()
  Associates an item in a select list with the type and output data buffer.
- OCIDefineByPos2()
  Associates an item in a select list with the type and output data buffer. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIStmtGetPieceInfo()
  Returns piece information for a piecewise operation.
- OCIStmtSetPieceInfo()
  Sets piece information for a piecewise operation.

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

> **See Also:**
>
> Runtime Data Allocation and Piecewise Operations in OCI for more information about `OCI_DYNAMIC_FETCH` mode

## 25.4.12 OCIDefineObject()

Sets up additional attributes necessary for a named data type or `REF` define.

**Purpose**

Sets up additional attributes necessary for a named data type or `REF` define.

**Syntax**

```
sword OCIDefineObject ( OCIDefine       *defnp,
                        OCIError        *errhp,
                        const OCIType   *type,
                        void           **pgvpp,
                        ub4             *pvszsp,
                        void           **indpp,
                        ub4             *indszp );
```

**Parameters**

**defnp (IN/OUT)**
A define handle previously allocated in a call to `OCIDefineByPos()` or `OCIDefineByPos2()`.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**type (IN) [optional]**
Points to the type descriptor object (TDO) that describes the type of the program variable. This parameter is optional for variables of type `SQLT_REF`, and may be passed as `NULL` if it is not being used.

**pgvpp (IN/OUT)**
Points to a pointer to a program variable buffer. For an array, `pgvpp` points to an array of pointers. Memory for the fetched named data type instances is dynamically allocated in the object cache. At the end of the fetch when all the values have been received, `pgvpp` points to the pointers to these newly allocated named data type instances. The application must call `OCIObjectFree()` to deallocate the named data type instances when they are no longer needed.

> **Note:**
>
> If the application wants the buffer to be implicitly allocated in the cache, `*pgvpp` should be passed in as `NULL`.

**pvszsp (IN/OUT)**
Points to the size of the program variable. For an array, it is an array of `ub4`.

**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 pointers to these newly allocated indicator structures.

**indszp (IN/OUT)**
Points to the sizes of the indicator structure program variable. For an array, it is an array of `ub4`s.

### Comments

This function follows a call to `OCIDefineByPos()` or `OCIDefineByPos2()` to set initial define information. This call sets up additional attributes necessary for a named data type define. An error is 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 data type `OCIType` for the named data type being defined. The TDO can be retrieved with a call to `OCIDescribeAny()`.

### Related Topics

- OCIDefineByPos()
  Associates an item in a select list with the type and output data buffer.
- OCIDefineByPos2()
  Associates an item in a select list with the type and output data buffer. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIObjectFree()
  Frees and unpins an object instance.
- OCIDescribeAny()
  Describes existing schema and subschema objects.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> - `OCIEnvCreate()`, and `OCIEnvNlsCreate()` for more information about initializing the OCI process environment
> - About Binding and Defining Multiple Buffers for an example of using multiple buffers

## 25.4.13 OCIDescribeAny()

Describes existing schema and subschema objects.

### Purpose

Describes existing schema and subschema objects.

### Syntax

```
sword OCIDescribeAny ( OCISvcCtx       *svchp,
                       OCIError        *errhp,
                       void            *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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 `hr.employees.department_id`, `hr` is the name of the schema, `employees` is the name of a table in the schema, and `department_id` 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. In addition, the `OCIDescribeAny()` call describes all package types and package type attributes contained in the package. This call also describes subschema objects, such as a column in a table. This call populates the describe handle with the object-specific attributes that 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.

By default, explicit describe (`OCIDescribeAny()`) does not list the invisible columns. To get the user defined invisible column's metadata, you must set the describe handle attribute `OCI_ATTR_SHOW_INVISIBLE_COLUMNS` before calling `OCIDescribeAny()`. To know whether the column is of an invisible type, you can get the column attribute `OCI_ATTR_INVISIBLE_COL` using `OCIAttrGet()`.

The property whether a column is visible or not can be controlled by the user. Invisible columns are not seen unless specified explicitly in the `SELECT` list. Any generic access of a table, such as a `SELECT * FROM` *`table-name`* statement or a `DESCRIBE` statement, will not show invisible columns.

### Related Topics

- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.
- OCIParamGet()
  Returns a descriptor of a parameter specified by position in the describe handle or statement handle.
- OCIAttrGet()
  Gets the value of an attribute of a handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> Describing Schema Metadata for more information about describe operations

## 25.4.14 OCIStmtGetBindInfo()

Gets the bind and indicator variable names.

### Purpose

Gets the bind and indicator variable names.

### Syntax

```
sword OCIStmtGetBindInfo ( OCIStmt      *stmtp,
                           OCIError     *errhp,
                           ub4           size,
                           ub4           startloc,
                           sb4          *found,
                           OraText      *bvnp[],
                           ub1           bvnl[],
                           OraText      *invp[],
                           ub1           inpl[],
                           ub1           dupl[],
                           OCIBind      *hndl[] );
```

### Parameters

**stmtp (IN)**
The statement handle prepared by `OCIStmtPrepare2()`.

**errhp (IN)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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)**
The expression `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. Is 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 a duplicate of another.

**hndl (OUT)**
An array that 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 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` is returned if the statement has no bind variables or if the starting bind position specified 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 `OCIStmtPrepare2()` prior to this call. The encoding setting in the statement handle determines whether Unicode strings are retrieved.

This call is processed locally.

### Related Topics

- OCIStmtPrepare2()
  Prepares a SQL or PL/SQL statement for execution.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 26

# More Oracle Database Access C API

This chapter describes more Oracle Database Access C API OCI relational functions.

This chapter describes more Oracle Database Access C API and completes the description of the OCI relational functions started in the previous 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 Database installation. For additional information, see OCI Demonstration Programs.

This chapter contains these topics:

- Introduction to the Relational Functions
- Statement Functions
- LOB Functions
- Database Advanced Queuing and Publish-Subscribe Functions
- Direct Path Loading Functions
- Thread Management Functions
- Transaction Functions
- Sharding Functions
- Miscellaneous Functions

## 26.1 Introduction to the Relational Functions

Introduces relational function calls.

This chapter describes the OCI relational function calls. This chapter and the previous one, cover the functions in the basic OCI.

> **See Also:**
>
> Error Handling in OCI for information about return codes and error handling

## 26.1.1 Conventions for OCI Functions

Describes conventions used for OCI functions.

See the Conventions for OCI Functions for the conventions used in describing each function.

# 26.2 Statement Functions

Lists and describes the statement functions.

Table 26-1 lists the statement functions that are described in this section. Use functions that end in "2" for all new applications.

**Table 26-1 Statement Functions**

| Function | Purpose |
|---|---|
| OCIStmtExecute() | Send statements to server for execution |
| OCIStmtFetch2() | Fetch rows from a query and fetches a row from the (scrollable) result set |
| OCIStmtGetNextResult() | Returns the implicit results from an executed PL/SQL statement handle |
| OCIStmtGetPieceInfo() | Get piece information for piecewise operations |
| OCIStmtPrepare2() | Prepare a SQL or PL/SQL statement for execution. The user also has the option of using the statement cache, if it has been enabled. |
| OCIStmtRelease() | Release the statement handle |
| OCIStmtSetPieceInfo() | Set piece information for piecewise operations |

## 26.2.1 OCIStmtExecute()

Associates an application request with a server.

### Purpose

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)**
A statement handle. It defines the statement and the associated data to be executed at the server. It is invalid to pass in a statement handle that has bind of data types only supported in release 8.x or later, when `svchp` points to an Oracle7 server.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information, when there is an error.

**iters (IN)**
For non-`SELECT` statements, the number of times this statement is executed equals `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 retrieves, then set `iters` to zero.
This function returns an error if `iters`=0 for non-`SELECT` statements.
This function returns an error if `iters` is not 1 for DDL 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 it is supplied, then it 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` statement. This facility allows multiple service contexts to Oracle Database 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 is optional. If it is supplied, then it must point to a descriptor of type `OCI_DTYPE_SNAP`. This descriptor is filled in with an opaque representation that is the current Oracle Database system change number (SCN) suitable as a `snap_in` input to a subsequent call to `OCIStmtExecute()`. To avoid "snapshot too old" errors, do not use this descriptor any longer than necessary.

**mode (IN)**
The modes are:

- `OCI_BATCH_ERRORS` - See Using Batch Error Mode 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 want to describe a query before execution. Calling `OCIStmtExecute()` in this mode does not execute the statement, but it does return the select-list description. To maximize performance, Oracle recommends that applications execute the statement in default mode and use the implicit describe that accompanies the execution.
- `OCI_EXACT_FETCH` - Used when the application knows in advance exactly how many rows it is fetching. This mode turns prefetching off for Oracle Database 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 before execution. Executing in this mode parses the query and returns parse errors in the SQL, if any. Users must note that this involves an additional round-trip to the server. To maximize performance, Oracle recommends that the user execute the statement in the default mode, which, parses the statement as part of the bundled operation.
- `OCI_STMT_SCROLLABLE_READONLY` - Required for the result set to be scrollable. The result set cannot be updated. See About Fetching Results for more information about this mode. This mode cannot be used with any other mode.

  `OCI_RETURN_ROW_COUNT_ARRAY` - This mode allows the user to get DML rowcounts per iteration. It is an error to pass this mode for statements that are not DMLs. See Statement Handle Attributes for more information. This mode can be used along with `OCI_BATCH_ERRORS`.

The modes are not mutually exclusive; you can use them 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, then 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:**
>
> About Describing Select-List Items

Also for `SELECT` statements, some results are available implicitly. Rows are 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. The 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 developer'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 canceled or an end of fetch condition is reached.

To reexecute a DDL statement, you must prepare the statement again using `OCIStmtPrepare2()`.

> **Note:**
>
> If output variables are defined for a `SELECT` statement before a call to `OCIStmtExecute()`, the number of rows specified by `iters` 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 OCIStmtFetch2() or deprecated OCIStmtFetch().

**Related Topics**

- OCIStmtPrepare2()
  Prepares a SQL or PL/SQL statement for execution.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> Polling Mode Operations in OCI

## 26.2.2 OCIStmtFetch2()

Fetches a row from the (scrollable) result set.

**Purpose**

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 that you can pass to `OCIErrorGet()` for diagnostic information if an error occurs.

**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 nonscrollable 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` - Fetches the row number (specified by `fetchOffset` parameter) in the result set using absolute positioning.
- `OCI_FETCH_RELATIVE` - Fetches 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 deprecated `OCIStmtFetch()` call, but with the addition of the `fetchOffset` parameter. It can be used on any statement handle, whether it is scrollable or not. For a *nonscrollable statement handle*, the only acceptable value of `orientation` is `OCI_FETCH_NEXT`, and the `fetchOffset` parameter is ignored.

For new applications you are encouraged to use this 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_UB8_ROW_COUNT` contains the highest absolute row value that was fetched.

All other orientation modes besides `OCI_FETCH_ABSOLUTE` and `OCI_FETCH_RELATIVE` ignore the `fetchOffset` value.

This call can also be used to determine 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. If `nrows` is set to be greater than 1 with `OCI_FETCH_LAST` orientation, nrows is considered to be 1.

The return codes are the same as for deprecated `OCIStmtFetch()`, except that `OER(1403)` with return code `OCI_NO_DATA` is 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 must be explicitly canceled (that is, fetch with 0 rows) or freed to release server-side resources for the scrollable cursor. A nonscrollable statement handle is implicitly canceled 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.

**Related Topics**

- OCIStmtFetch()
  This function was deprecated in a release previous to Oracle 11*g* R2 (11.2).
- OCIStmtExecute()
  Associates an application request with a server.
- OCIBindByPos()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIBindByPos2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- About Using Scrollable Cursors in OCI
  A cursor is a current position in a *result set*.
- Polling Mode Operations in OCI
  OCI has calls that poll for completion.

## 26.2.3 OCIStmtGetNextResult()

Returns the implicit results from an executed PL/SQL statement handle.

**Purpose**

Returns the implicit results from an executed PL/SQL statement handle.

### Syntax

```
sword OCIStmtGetNextResult (OCIStmt   *stmthp,
                            OCIError  *errhp,
                            void     **result,
                            ub4       *rtype,
                            ub4        mode)
```

### Parameters

**stmthp (IN)**
The executed statement handle.

**errhp (IN)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**result (OUT)**
The next implicit result from the executed PL/SQL statement.

**rtype (OUT)**
The type of the implicit result. The only possible value is `OCI_RESULT_TYPE_SELECT`.

**mode (IN)**
The only possible value is `OCI_DEFAULT` (default mode).

### Comments

Each call to `OCIStmtGetNextResult()` retrieves a single implicit result in the order in which they were returned from the PL/SQL procedure or block. If no more results are available, then `OCI_NO_DATA` is returned. If `rtype` is `OCI_RESULT_TYPE_SELECT`, then the returned result can be cast as an OCI statement handle, and is allocated by OCI. Applications can do normal OCI define and fetch calls to fetch rows from the implicit result sets. The returned OCI statement handle cannot be freed explicitly. All implicit result sets are automatically closed and freed when the top-level statement handle is freed or released.

See OCI_ATTR_IMPLICIT_RESULT_COUNT for information about this statement handle attribute, which returns the total number of implicit results available on the top-level OCI statement handle.

### Returns

Returns one of the following:

- `OCI_ERROR`
- `OCI_SUCCESS`
- `OCI_NO_DATA` - When all implicit result sets have been retrieved from the top-level statement handle

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.2.4 OCIStmtGetPieceInfo()

Returns piece information for a piecewise operation.

### Purpose

Returns piece information for a piecewise operation.

### Syntax

```
sword OCIStmtGetPieceInfo( const OCIStmt  *stmtp,
                           OCIError       *errhp,
                           void          **hndlpp,
                           ub4            *typep,
                           ub1            *in_outp,
                           ub4            *iterp,
                           ub4            *idxp,
                           ub1            *piecep );
```

### Parameters

**stmtp (IN)**
The statement that when executed returned `OCI_NEED_DATA`.

**errhp (OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**hndlpp (OUT)**
Returns a pointer to the bind or define handle of the bind or define whose run-time 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 these defined values: `OCI_ONE_PIECE`, `OCI_FIRST_PIECE`, `OCI_NEXT_PIECE`, or `OCI_LAST_PIECE`.

### Comments

When an execute or fetch call returns `OCI_NEED_DATA` to get or return a dynamic bind, define value, or piece, `OCIStmtGetPieceInfo()` returns the relevant information: bind or define handle, iteration, index number, and which piece.

**Related Topics**

- OCIArrayDescriptorAlloc()
  Allocates an array of descriptors.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIStmtExecute()
  Associates an application request with a server.
- OCIStmtFetch2()
  Fetches a row from the (scrollable) result set.
- OCIStmtSetPieceInfo()
  Sets piece information for a piecewise operation.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Polling Mode Operations in OCI
  OCI has calls that poll for completion.

> **See Also:**
>
> - Runtime Data Allocation and Piecewise Operations in OCI for more information about using `OCIStmtGetPieceInfo()`

## 26.2.5 OCIStmtPrepare2()

Prepares a SQL or PL/SQL statement for execution.

**Purpose**

Prepares a SQL or PL/SQL statement for execution. The user has the option of using the statement cache, if it has been enabled.

Oracle strongly encourages use of `OCIStmtPrepare2()` in all OCI applications instead of the deprecated call `OCIStmtPrepare()`.

**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.

**stmthp (OUT)**
Pointer to the statement handle returned.

**errhp (IN)**
A pointer to the error handle for diagnostics.

**stmttext (IN)**
The statement text. The semantics of the `stmttext` are same as those 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 be used for searching the statement in the statement cache. 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 as follows:

- `OCI_V7_SYNTAX` - V7 ORACLE parsing syntax.
- `OCI_NTV_SYNTAX` - Syntax depends upon the version of the server.

  `OCI_FOREIGN_SYNTAX` - Specifies the statement to be translated according to the SQL translation profile set in the session.

**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 have caching enabled, and if caching is not enabled, then all statements are not cached.
The valid modes are as follows:

- `OCI_DEFAULT` - Caching is not enabled. If the statement is not found in the cache, this mode allocates a new statement handle and prepares the statement handle for execution. If the statement is not found in the cache and one of the following circumstances applies, then the subsequent actions follow:
  - Only the text has been supplied: a new statement is allocated and prepared and returned. The tag `NULL`. `OCI_SUCCESS` is returned.
  - Only the tag has been supplied: `stmthp` is `NULL`. `OCI_ERROR` is returned.
  - Both text and key were supplied: a new statement is allocated and prepared and returned. The tag `NULL`. `OCI_SUCCESS_WITH_INFO` is 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` is returned. Otherwise, `OCI_ERROR` is 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` is the return status from the execution. Use `OCIErrorGet()` to find the new error number corresponding to the warnings.
- `OCI_PREP2_IMPL_RESULTS_CLIENT` - The mode should be passed as `OCI_PREP2_IMPL_RESULTS_CLIENT` when this call is made in an external procedure and implicit results need to be processed. See OCI Support for Implicit Results for more details.

**Related Topics**

- OCIStmtPrepare()
  This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).
- OCIStmtRelease()
  Releases the statement handle obtained by a call to `OCIStmtPrepare2()`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.2.6 OCIStmtRelease()

Releases the statement handle obtained by a call to `OCIStmtPrepare2()`.

**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 is a SQL string passed in by the caller. If a `NULL` key is passed in, the statement is not 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 is not kept in the cache anymore.

**Related Topics**

- OCIStmtPrepare2()
  Prepares a SQL or PL/SQL statement for execution.

## 26.2.7 OCIStmtSetPieceInfo()

Sets piece information for a piecewise operation.

### Purpose

Sets piece information for a piecewise operation.

### Syntax

```
sword OCIStmtSetPieceInfo ( void             *hndlp,
                            ub4               type,
                            OCIError         *errhp,
                            const void       *bufp,
                            ub4              *alenp,
                            ub1               piece,
                            const void       *indp,
                            ub2              *rcodep );
```

### Parameters

**hndlp (IN/OUT)**
The bind or define handle.

**type (IN)**
Type of the handle.

**errhp (OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**bufp (IN/OUT)**
A pointer to storage containing the data value or the piece when it is an IN bind variable; otherwise, `bufp` is a pointer to storage for getting a piece or a value for OUT binds and define variables. For named data types or `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 are:

- `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 an `sb2` value or pointer to an indicator structure for named data types (SQLT_NTY) and `REF`s (SQLT_REF), that is, depending upon the data type, `*indp` is either an `sb2` or a `void *`.

**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.

### Related Topics

- OCIArrayDescriptorAlloc()
  Allocates an array of descriptors.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIStmtExecute()
  Associates an application request with a server.
- OCIStmtFetch2()
  Fetches a row from the (scrollable) result set.
- OCIStmtGetPieceInfo()
  Returns piece information for a piecewise operation.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Polling Mode Operations in OCI
  OCI has calls that poll for completion.

> **See Also:**
>
> - Runtime Data Allocation and Piecewise Operations in OCI for more information about using `OCIStmtSetPieceInfo()`

## 26.3 LOB Functions

Table 26-2 lists the LOB functions that use the LOB locator that are described in this section. Use functions that end in "2" for all new applications.

Beginning with Oracle Database 12c Release 2 (12.2), most LOB functions support Application Continuity (see When Application Continuity in OCI Can Fail Over for a complete list). This support also includes some deprecated LOB functions for compatibility listed in Deprecated Lob Functions.

Beginning with Oracle Database 12c Release 2 (12.2), OCI LOB functions support distributed operations on remote LOBs (CLOB, BLOB) except for OCI LOB functions for BFILES. This includes support for certain queries that select a remote LOB locator that in previous releases returned an error. The only restriction is that all LOB functions that take in two locators should have both LOBs collocated at one database. The following LOB functions throw an error when a remote locator is passed to it: `OCILobAssign()`, `OCILobLocatorAssign()`, `OCILobEnableBuffering()`, `OCILobDisableBuffering()`, `OCILobFlushBuffer()`, `OCILobArrayRead()`, `OCILobArrayWrite()`, and `OCILobLoadFromFile2()`.

**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 locations:

- About Binding LOB Data for usage and examples for both `INSERT` and `UPDATE` statements
- About Defining LOB Data for usage and examples of `SELECT` statements
- LOB and BFILE Operations

**Table 26-2 LOB Functions**

| Function | Purpose |
|---|---|
| OCIDurationBegin() | Start user duration for temporary LOB |
| OCIDurationEnd() | End user duration for temporary LOB |
| OCILobAppend() | Append one LOB to another |
| OCILobArrayRead() | Read LOB data for multiple locators |
| OCILobArrayWrite() | Write LOB data for multiple locators |
| OCILobAssign() | Assign one LOB locator to another |
| OCILobCharSetForm() | Get character set form from LOB locator |
| OCILobCharSetId() | Get character set ID from LOB locator |
| OCILobClose() | Close a previously opened LOB |
| OCILobCopy2() | Copy all or part of one LOB to another. This function must be used for LOBs of size greater than 4 GB. |
| OCILobCreateTemporary() | Create a temporary LOB |
| OCILobDisableBuffering() | Turn LOB buffering off |
| OCILobEnableBuffering() | Turn LOB buffering on |
| OCILobErase2() | Erase a portion of a LOB. This function must be used for LOBs of size greater than 4 GB. |
| OCILobFileClose() | Close a previously opened `BFILE` |
| OCILobFileCloseAll() | Close all previously opened files |
| OCILobFileExists() | Check if a file exists on the server |

**Table 26-2 (Cont.) LOB Functions**

| Function | Purpose |
|---|---|
| OCILobFileGetName() | Get directory object and file name from the LOB locator |
| OCILobFileIsOpen() | Check if file on server is open using this locator |
| OCILobFileOpen() | Open a `BFILE` |
| OCILobFileSetName() | Set directory object and file name in the LOB locator |
| OCILobFlushBuffer() | Flush the LOB buffer |
| OCILobFreeTemporary() | Free a temporary LOB |
| OCILobGetChunkSize() | Get the chunk size of a LOB |
| OCILobGetContentType() | Retrieve the user-specified content type string (a file format identifier) for a SecureFile |
| OCILobGetLength2() | Get length of a LOB. This function must be used for LOBs of size greater than 4 GB. |
| OCILobGetOptions() | Get option settings of a SecureFile |
| OCILobGetStorageLimit() | Get the maximum length of an internal LOB (BLOB, CLOB, or NCLOB) in bytes |
| OCILobIsEqual() | Compare two LOB locators for Equality |
| OCILobIsOpen() | Check to see if a LOB is open |
| OCILobIsTemporary() | Determine if a given LOB is temporary |
| OCILobLoadFromFile2() | Load a LOB from a `BFILE`. This function must be used for LOBs of size greater than 4 GB. |
| OCILobLocatorAssign() | Assign one LOB locator to another |
| OCILobLocatorIsInit() | Check to see if a LOB locator is initialized |
| OCILobOpen() | Open a LOB |
| OCILobRead2() | Read a portion of a LOB. This function must be used for LOBs of size greater than 4 GB. |
| OCILobSetContentType() | Store the user-specified content type string of the SecureFile |
| OCILobSetOptions() | Enable option settings for existing and newly created SecureFiles |
| OCILobTrim2() | Truncate a LOB. This function must be used for LOBs of size greater than 4 GB. |
| OCILobWrite2() | Write into a LOB. This function must be used for LOBs of size greater than 4 GB. |
| OCILobWriteAppend2() | Write data beginning at the end of a LOB. This function must be used for LOBs of size greater than 4 GB. |

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 `CLOB`s and `NCLOB`s, and in bytes for `BLOB`s and `BFILE`s.
- 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()` (deprecated), `OCILobGetLength2()`, `OCILobRead()` (deprecated), `OCILobRead2()`, `OCILobWrite()` (deprecated), and `OCILobWrite2()`.
  - Offset (`offset`) parameter - Regardless of whether the client-side character set is varying-width, the offset parameter is always in characters for `CLOB`s and `NCLOB`s and in bytes for `BLOB`s and `BFILE`s.
- For many of the LOB operations, regardless of the client-side character set, the amount parameter is in characters for `CLOB`s and `NCLOB`s. These LOB operations include `OCILobCopy2()`, `OCILobErase2()`, `OCILobGetLength2()`, `OCILobLoadFromFile2()`, and `OCILobTrim2()`. 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.

## 26.3.1 OCIDurationBegin()

Starts a user duration for a temporary LOB.

### 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. It is 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 `duration` parameter is used to return a number that uniquely identifies the duration created by this call.

### Related Topics

- Temporary LOB Durations
  OCI supports several predefined durations for temporary LOBs, and a set of functions that the application can use to define application-specific durations.

## 26.3.2 OCIDurationEnd()

Terminates a user duration for a temporary LOB.

### 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.

### Related Topics

- OCIDurationBegin()
  Starts a user duration for a temporary LOB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Temporary LOB Durations
  OCI supports several predefined durations for temporary LOBs, and a set of functions that the application can use to define application-specific durations.

**Related Topics**

- OCIDurationBegin()
  Starts a user duration.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.3 OCILobAppend()

Appends a LOB value at the end of another LOB as specified.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**dst_locp (IN/OUT)**
An internal LOB locator uniquely referencing the destination LOB. This locator must have been 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 have been 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 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 (GB)) 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 `BLOB`s or both be `CLOB`s). 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 before 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 before performing this operation, then you must close it before you commit 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

**Related Topics**

- OCILobTrim2()
  Truncates the LOB value to a shorter length. This function must be used for LOBs of size greater than 4 GB.
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.4 OCILobArrayRead()

Reads LOB data for multiple locators in one round-trip.

**Purpose**

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,
                        void                    **bufp_arr,
                        oraub8                    bufl_arr,
                        ub1                       piece,
                        void                     *ctxp,
                        OCICallbackLobArrayRead  (cbfp)
                                (
                                 void           *ctxp,
                                 ub4             array_iter,
                                 const void     *bufp,
                                 oraub8          lenp,
                                 ub1             piecep
                                 void          **changed_bufpp,
                                 oraub8         *changed_lenp
                                )
```

```
                         ub2                    csid,
                         ub1                    csfrm );
```

**Parameters**

**svchp (IN/OUT)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 - In polling mode, this parameter 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 (`CLOB`s, `NCLOB`s) it is the number of characters from the beginning of the LOB; for binary LOBs or `BFILE`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 piece is 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 can be registered to be called for each piece. If this is `NULL`, then `OCI_NEED_DATA` is 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.
The `csfrm` parameter 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.
>
> `OCILobArrayRead()` throws an error when a remote locator is passed to it.

For `BFILE`s, the operating system file must exist on the server, and it must have been opened by `OCILobFileOpen()` or `OCILobOpen()` using the input locator. The Oracle Database must have permission to read the operating system file, and the user must have read permission on the directory object.

When you use the polling mode for `OCILobArrayRead()`, the first call must specify values for `offset_arr` and `amt_arr`, but on subsequent polling calls to `OCILobArrayRead()`, you 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 you use 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 `CLOB`s and `NCLOB`s, `char_amt_arr` gives the number of characters read in the current piece.
- When you use 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 you use polling, examine the `byte_amt_arr` parameter to see how much the buffer is filled for the current piece. For `CLOB`s and `NCLOB`s, `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.

**Related Topics**

- OCILobFileOpen()
  Opens a `BFILE` on the file system of the server for read-only access.
- OCILobOpen()
  Opens a LOB, internal or external, in the indicated mode.

- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobFileSetName()
  Sets the directory object and file name in the `BFILE` locator.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobArrayWrite()
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> - PL/SQL REF CURSORs and Nested Tables in OCI for additional information on Unicode format
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s
> - The demonstration programs included with your Oracle Database installation for a code sample showing the use of LOB reads and writes.
> - OCI Demonstration Programs
> - Runtime Data Allocation and Piecewise Operations in OCI for general information about piecewise OCI operations

## 26.3.5 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,
                         void                 **bufp_arr,
                         oraub8                *bufl_arr,
                         ub1                    piece,
                         void                  *ctxp,
                         OCICallbackLobArrayWrite (cbfp)
                                  (
                                    void     *ctxp,
                                    ub4       array_iter,
                                    void     *bufp,
                                    oraub8   *lenp,
                                        ub1      *piecep
                                        void    **changed_bufpp,
                                        oraub8   *changed_lenp
                                      )
                   ub2                      csid,
                   ub1                      csfrm );
```

**Parameters**

**svchp (IN/OUT)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 - In polling mode this parameter 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 are 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 that the buffer is 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 can be registered to be called for each piece. If this is `NULL`, then `OCI_NEED_DATA` is 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 the 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 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 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.
The `csfrm` parameter 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 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 the buffers can be provided piecewise using callbacks or a standard polling method.

`OCILobArrayWrite()` throws an error when a remote locator is passed to it.

> **Note:**
>
> When you read or write LOBs, specify a character set form (`csfrm`) that matches the form of the locator itself.
>
> The parameters `piece`, `csid`, and `csfrm` are the same for all locators of the array.

When you use the polling mode for `OCILobArrayWrite()`, the first call must specify values for `offset_arr`, `byte_amt_arr`, and `char_amt_arr`, but on subsequent polling calls to `OCILobArrayWrite()`, you 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 is invoked to get the next piece after a piece is written to the pipe. Each piece is 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. `BFILE`s are not valid, because 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 `CLOB`s and `NCLOB`s, `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 `BLOB`s, `byte_amt_arr` returns the number of bytes written by each piece, whereas `char_amt_arr` is undefined on output.

To write 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.

It is not mandatory that you wrap this LOB operation inside the open or close calls. If you did not open the LOB before 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 before performing this operation, then you must close it before you commit 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

**Related Topics**

- OCILobRead2()
  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.
- OCILobAppend()
  Appends a LOB value at the end of another LOB as specified.
- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobArrayRead()
  Reads LOB data for multiple locators in one round-trip.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

**See Also:**

- PL/SQL REF CURSORs and Nested Tables in OCI for additional information on Unicode format
- The demonstration programs included with your Oracle Database installation for a code sample showing the use of LOB reads and writes.
- OCI Demonstration Programs
- Runtime Data Allocation and Piecewise Operations in OCI for general information about piecewise OCI operations

## 26.3.6 OCILobAssign()

Assigns one LOB or `BFILE` locator to another

**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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 copies the LOB value.

`OCILobAssign()` throws an error when a remote locator is passed to it.

`OCILobAssign()` cannot be used for temporary LOBs; it generates an `OCI_INVALID_HANDLE` error. For temporary LOBs, use `OCILobLocatorAssign()`.

For `BFILE`s, 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 can modify the LOB data through the LOB buffering subsystem.

The value of the input destination locator must have been allocated with a call to `OCIDescriptorAlloc()`. For example, assume the following declarations:

```
OCILobLocator     *source_loc = (OCILobLocator *) 0;
OCILobLocator     *dest_loc = (OCILobLocator *) 0;
```

The following code example shows how an application could allocate the `source_loc` locator.

Allocating a source_loc Source Locator

```
if (OCIDescriptorAlloc((void  *) envhp, (void  **) &source_loc,
    (ub4) OCI_DTYPE_LOB, (size_t) 0, (void  **) 0))
    handle_error;
```

Assume that it then selects a LOB from a table into the `source_loc` 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`, as shown in the following code example.

Allocating a dest_loc Destination Locator

```
if (OCIDescriptorAlloc((void  *) envhp, (void  **) &dest_loc,
    (ub4)OCI_DTYPE_LOB, (size_t) 0, (void  **) 0))
    handle_error;
if (OCILobAssign(envhp, errhp, source_loc, &dest_loc))
    handle_error
```

**Related Topics**

- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.
- OCILobIsEqual()
  Compares two LOB or `BFILE` locators for equality.
- OCILobLocatorAssign()
  Assigns one LOB or `BFILE` locator to another.
- OCILobLocatorIsInit()
  Tests to see if a given LOB or `BFILE` locator is initialized.
- OCILobEnableBuffering()
  This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.7 OCILobCharSetForm()

Gets the character set form of the LOB locator, if any.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 because there is no concept of a character set for binary LOBs and `BFILE`s. The caller must allocate space for `csfrm` (a `ub1`).
The `csfrm` parameter 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 Topics

- OCILobCharSetId()
  Gets the LOB locator's database character set ID of the LOB locator, if any.
- OCILobLocatorIsInit()
  Tests to see if a given LOB or `BFILE` locator is initialized.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.8 OCILobCharSetId()

Gets the LOB locator's database character set ID of the LOB locator, if any.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 because there is no concept of a character set for binary LOBs or binary 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 Topics**

- OCILobCharSetForm()
  Gets the character set form of the LOB locator, if any.
- OCILobLocatorIsInit()
  Tests to see if a given LOB or `BFILE` locator is initialized.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.9 OCILobClose()

Closes a previously opened LOB or `BFILE`.

#### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 triggers 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 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, 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

**Related Topics**

- OCILobOpen()
  Opens a LOB, internal or external, in the indicated mode.
- OCILobIsOpen()
  Tests whether a LOB or `BFILE` is open.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Functions for Opening and Closing LOBs
  OCI provides functions to explicitly open a LOB, `OCILobOpen()`, to close a LOB, `OCILobClose()`, and to test whether a LOB is open, `OCILobIsOpen()`.

## 26.3.10 OCILobCopy2()

Copies all or a portion of a LOB value into another LOB value.

### 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

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**dst_locp (IN/OUT)**
An internal LOB locator uniquely referencing the destination LOB. This locator must have been 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 have been a locator that was obtained from the server specified by `svchp`.

**amount (IN)**
The number of characters for `CLOB`s or `NCLOB`s or the number of bytes for `BLOB`s to be copied from the source LOB to the destination LOB. The maximum value accepted by this parameter is `UB8MAXVAL` (18446744073709551615). Specifying `UB8MAXVAL` also indicates that the entire source LOB will be copied to the destination LOB using the specified source and destination offsets.

**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 exist.

Copying a complete SecureFile in a column or partition with deduplicate enabled, to a LOB in the same column or partition, causes the copy to be deduplicated.

If the data 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 `BLOB`s) or spaces (for `CLOB`s) 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. Use OCILobCopy2() for LOBs of size greater than 4 GB.

Both the source and the destination LOB locators must be of the same type (that is, they must both be `BLOB`s or both be `CLOB`s). 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 before 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 before performing this operation, then you must close it before you commit 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

> **Note:**
>
> You can call OCILobGetLength2() to determine the length of the source LOB.

**Related Topics**

- OCILobRead2()
  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.
- OCILobAppend()
  Appends a LOB value at the end of another LOB as specified.
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.11 OCILobCreateTemporary()

Creates a temporary LOB.

**Purpose**

Creates 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN/OUT)**
A locator that points to the temporary LOB. You must allocate the locator using `OCIDescriptorAlloc()` before passing it to this function. It does not matter whether this locator points to a LOB; the temporary LOB gets 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. The `csfrm` parameter 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; pass `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.

**Related Topics**

- OCILobFreeTemporary()
  Frees a temporary LOB.
- OCILobIsTemporary()
  Tests if a locator points to a temporary LOB.
- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> Temporary LOB Support for more information about temporary LOBs and their durations

## 26.3.12 OCILobErase2()

Erases a specified portion of the internal LOB data starting at a specified offset.

**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**

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN/OUT)**
An internal LOB locator that uniquely references the LOB. This locator must have been 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 before 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 before performing this operation, then you must close it before you commit 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

### Related Topics

- OCILobRead2()
  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.
- OCILobAppend()
  Appends a LOB value at the end of another LOB as specified.
- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.13 OCILobFileClose()

Closes a previously opened `BFILE`.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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.

**Related Topics**

- OCILobClose()
  Closes a previously opened LOB or `BFILE`.
- OCILobFileCloseAll()
  Closes all open `BFILE`s on a given service context.
- OCILobFileExists()
  Tests to see if the `BFILE` exists on the server's operating system.
- OCILobFileIsOpen()
  Tests to see if the `BFILE` is open.
- OCILobFileOpen()
  Opens a `BFILE` on the file system of the server for read-only access.
- OCILobOpen()
  Opens a LOB, internal or external, in the indicated mode.
- OCILobIsOpen()
  Tests whether a LOB or `BFILE` is open.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s

## 26.3.14 OCILobFileCloseAll()

Closes all open `BFILE`s on a given service context.

### Purpose

Closes all open `BFILE`s 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

### Comments

Closes all open `BFILE`s on a given service context.

**Related Topics**

- OCILobFileClose()
  Closes a previously opened `BFILE`.
- OCILobFileExists()
  Tests to see if the `BFILE` exists on the server's operating system.
- OCILobFileIsOpen()
  Tests to see if the `BFILE` is open.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s

## 26.3.15 OCILobFileExists()

Tests to see if the `BFILE` exists on the server's operating system.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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.

### Related Topics

- OCILobFileClose()
  Closes a previously opened `BFILE`.
- OCILobFileCloseAll()
  Closes all open `BFILE`s on a given service context.
- OCILobFileIsOpen()
  Tests to see if the `BFILE` is open.
- OCILobOpen()
  Opens a LOB, internal or external, in the indicated mode.
- OCILobIsOpen()
  Tests whether a LOB or `BFILE` is open.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s

## 26.3.16 OCILobFileGetName()

Gets the `BFILE` locator's directory object and file name.

### Purpose

Gets the `BFILE` locator's directory object and file name.

### Syntax

```
sword OCILobFileGetName ( OCIEnv                  *envhp,
                          OCIError                *errhp,
                          const OCILobLocator     *filep,
                          OraText                 *dir_alias,
                          ub2                     *d_length,
                          OraText                 *filename,
                          ub2                     *f_length );
```

### Parameters

**envhp (IN/OUT)**
OCI environment handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 code point 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 it is in Unicode. It is an error to call this function for an internal LOB.

### Related Topics

- OCILobFileSetName()
  Sets the directory object and file name in the `BFILE` locator.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s

## 26.3.17 OCILobFileIsOpen()

Tests to see if the `BFILE` is open.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**filep (IN)**
Pointer to the `BFILE` locator being examined.

**flag (OUT)**
Returns `TRUE` if the `BFILE` was opened using this particular locator; returns `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.

**Related Topics**

- OCILobClose()
  Closes a previously opened LOB or `BFILE`.
- OCILobFileCloseAll()
  Closes all open `BFILE`s on a given service context.
- OCILobFileExists()
  Tests to see if the `BFILE` exists on the server's operating system.
- OCILobFileClose()
  Closes a previously opened `BFILE`.
- OCILobFileIsOpen()
  Tests to see if the `BFILE` is open.
- OCILobFileOpen()
  Opens a `BFILE` on the file system of the server for read-only access.
- OCILobOpen()
  Opens a LOB, internal or external, in the indicated mode.
- OCILobIsOpen()
  Tests whether a LOB or `BFILE` is open.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s

## 26.3.18 OCILobFileOpen()

Opens a `BFILE` on the file system of the server for read-only access.

**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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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. `BFILE`s can not be written through Oracle Database. 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.

### Related Topics

- OCILobClose()
  Closes a previously opened LOB or `BFILE`.
- OCILobFileCloseAll()
  Closes all open `BFILE`s on a given service context.
- OCILobFileExists()
  Tests to see if the `BFILE` exists on the server's operating system.
- OCILobFileClose()
  Closes a previously opened `BFILE`.
- OCILobFileIsOpen()
  Tests to see if the `BFILE` is open.
- OCILobFileOpen()
  Opens a `BFILE` on the file system of the server for read-only access.
- OCILobOpen()
  Opens a LOB, internal or external, in the indicated mode.
- OCILobIsOpen()
  Tests whether a LOB or `BFILE` is open.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s

## 26.3.19 OCILobFileSetName()

Sets the directory object and file name in the `BFILE` locator.

### Purpose

Sets the directory object and file name in the `BFILE` locator.

### Syntax

```
sword OCILobFileSetName ( OCIEnv            *envhp,
                          OCIError          *errhp,
                          OCILobLocator    **filepp,
                          const OraText     *dir_alias,
                          ub2                d_length,
                          const OraText     *filename,
                          ub2                f_length );
```

### Parameters

**envhp (IN/OUT)**
OCI environment handle. Contains the UTF-16 setting.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 (in bytes) of the input `dir_alias` parameter.

**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 (in bytes) of the input `filename` parameter.

### Comments

It is an error to call this function for an internal LOB.

### Related Topics

- OCILobFileGetName()
  Gets the `BFILE` locator's directory object and file name.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s

## 26.3.20 OCILobFreeTemporary()

Frees a temporary LOB.

### Purpose

Frees 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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. You must always call `OCILobFreeTemporary()` before calling `OCIDescriptorFree()` or `OCIArrayDescriptorFree()` to free the contents of the temporary LOB. See About Freeing Temporary LOBs for more information.

This function returns an error if the LOB locator passed in the `locp` parameter does not point to a temporary LOB, possibly because the LOB locator:

- Points to a permanent LOB
- Pointed to a temporary LOB that has been freed
- Has never pointed to anything

### Related Topics

- OCILobCreateTemporary()
  Creates a temporary LOB.
- OCILobIsTemporary()
  Tests if a locator points to a temporary LOB.

- OCIDescriptorFree()
  Deallocates a previously allocated descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.21 OCILobGetChunkSize()

Gets the chunk size of a LOB.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN/OUT)**
The internal LOB for which to get the usable chunk size.

**chunk_size (OUT)**
For LOBs with storage parameter `BASICFILE`, 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.
The `chunk_size` parameter is returned in terms of bytes for `BLOB`s, `CLOB`s, and `NCLOB`s.
For LOBs with storage parameter `SECUREFILE`, `chunk_size` is an advisory size and is provided for backward compatibility.

### Comments

When creating a table that contains an internal LOB, the user can specify the chunking factor, which can be a multiple of Oracle Database 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 is improved if the application issues read or write requests using a multiple of this chunk size. For writes, there is an added benefit because LOB chunks are versioned and, if all writes are done on a chunk basis, no extra versioning is done or duplicated. Users could batch up the write until they have enough for a chunk instead of issuing several write calls for the same chunk.

**Related Topics**

- OCILobGetStorageLimit()
  Gets the maximum length of an internal LOB (`BLOB`, `CLOB`, or `NCLOB`) in bytes.
- OCILobRead2()
  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.
- OCILobAppend()
  Appends a LOB value at the end of another LOB as specified.
- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> About Improving LOB Read/Write Performance

## 26.3.22 OCILobGetContentType()

Gets the user-specified content type string for the data in a SecureFile, if set.

**Purpose**

Gets the user-specified content type string for the data in a SecureFile, if set.

**Syntax**

```
sword OCILobGetContentType ( OCIEnv         *envhp,
                             OCISvcCtx      *svchp,
                             OCIError       *errhp,
                             OCILobLocator  *lobp,
                             oratext        *contenttypep,
                             ub4            *contenttypelenp,
                             ub4             mode );
```

**Parameters**

**envhp (IN)**
The environment handle.

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that can be passed to `OCIErrorGet()` for diagnostic information when there is an error.

**lobp (IN)**
A LOB locator that uniquely references a LOB.

**contenttypep (IN/OUT)**
Pointer to the buffer where the content type is stored after successful execution. You must allocate the buffer before calling this function. The size of the allocated buffer must be >= `OCI_LOB_CONTENTTYPE_MAXSIZE`.

**contenttypelenp (IN/OUT)**
Set this field to the size of `contenttypep` buffer. After the call successfully executes, this field holds the size of the `contenttypep` returned.

**mode (IN)**
For future use. Pass zero now.

### Comments

This function only works on SecureFiles. If `lobp` is not a SecureFile, then the error `SECUREFILE_WRONGTYPE` is returned. If `lobp` is buffered, a temporary LOB, or an abstract LOB, then the error `SECUREFILE_BADLOB` is returned.

If the SecureFile does not have a `contenttype` associated with it, the `contenttype` length (`contenttypelenp`) is returned as 0 without modifying the buffer `contenttypep`.

The maximum possible size of the `ContentType` string is defined as:

```
#define OCI_LOB_CONTENTTYPE_MAXSIZE 128
```

The `ContentType` is ASCII (that is, 1-byte/7-bit UTF8).

### Related Topics

- OCILobSetContentType()
  Sets a content type string for the data in the SecureFile to something that can be used by an application.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.23 OCILobGetLength2()

Gets the length of a LOB. This function must be used for LOBs of size greater than 4 GB.

### Purpose

You can also use this function for LOBs smaller than 4 GB.

### Syntax

```
sword OCILobGetLength2 ( OCISvcCtx      *svchp,
                         OCIError       *errhp,
                         OCILobLocator  *locp,
                         oraub8         *lenp );
```

### Parameters

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN)**
A LOB locator that uniquely references the LOB. For internal LOBs, this locator must have been 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 `OCILobErase2()` or `OCILobWrite2()` are also included in the length count.

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.
- OCILobFileSetName()
  Sets the directory object and file name in the `BFILE` locator.
- OCILobRead2()
  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.
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.
- OCILobAppend()
  Appends a LOB value at the end of another LOB as specified.

- OCILobLoadFromFile2()
  Loads and copies all or a portion of the file into an internal LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.24 OCILobGetOptions()

Obtains the enabled settings corresponding to the given input option types for a given SecureFile LOB.

### Purpose

Obtains the enabled settings corresponding to the given input option types for a given SecureFile LOB.

### Syntax

```
sword OCILobGetOptions ( OCISvcCtx         *svchp,
                         OCIError          *errhp,
                         OCILobLocator     *locp,
                         ub4                option_types,
                         void              *optionsp,
                         ub4                optionslenp,
                         ub4                mode );
```

### Parameters

**svchp (IN/OUT)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN/OUT)**
The LOB locator or `BFILE` locator that uniquely references the LOB or `BFILE`. This locator must have been obtained from the server specified by `svchp`.

**option_types (IN)**
The given option types that can be combined by a bit-wise inclusive `OR` (symbol "|"):

- Compression - `OCI_LOB_OPT_COMPRESS`
- Encryption - `OCI_LOB_OPT_ENCRYPT`
- Deduplication - `OCI_LOB_OPT_DEDUPLICATE`

**optionsp (OUT)**
The current settings for each of the option types given. Possible values are:

- `OCI_LOB_OPT_COMPRESS_ON`

- `OCI_LOB_OPT_ENCRYPT_ON`
- `OCI_LOB_OPT_DEDUPLICATE_ON`

**optionslenp (OUT)**
The length of the value in `optionsp`.

**mode (IN)**
Reserved for future use. Pass in 0.

### Comments

You can only specify option types that have been enabled on the column. An error is returned when an attempt is made to get the value of an option type that is not enabled on the column. For example, if you have a LOB column with compression enabled, and you call `OCILobGetOptions()` with `OCI_LOB_OPT_ENCRYPT` set in the `option_types` parameter, an error occurs.

Note that the returned value is a `ub4` pointer cast as a `void` pointer to allow for future expansion of option types and values. The `optionslenp` returned should be equal to `sizeof(ub4)`.

**Related Topics**

- OCILobSetOptions()
  Enables option settings for a SecureFile LOB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.25 OCILobGetStorageLimit()

Gets the maximum length of an internal LOB (`BLOB`, `CLOB`, or `NCLOB`) in bytes.

### 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 `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN)**
A LOB locator that uniquely references the LOB. The locator must have been 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 2 KB to 32 KB, the maximum LOB size ranges from 8 terabytes to 128 terabytes (TB) for LOBs.

### Related Topics

- OCILobGetChunkSize()
  Gets the chunk size of a LOB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- About Using LOBs of Size Greater than 4 GB
  Starting with Oracle Database 10*g* 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.

## 26.3.26 OCILobIsEqual()

Compares two LOB or `BFILE` locators for equality.

### 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 Topics**

- OCILobAssign()
  Assigns one LOB or `BFILE` locator to another
- OCILobLocatorIsInit()
  Tests to see if a given LOB or `BFILE` locator is initialized.

## 26.3.27 OCILobIsOpen()

Tests whether a LOB or `BFILE` is open.

### 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 that can be passed to `OCIErrorGet()` for diagnostic information when there is 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 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`. Multiple opens can be performed on the same `BFILE` using different locators. In other words, openness is associated with a specific locator for `BFILE`s.

**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 open. For external LOBs (`BFILE`s), this call also

requires a round-trip because the operating system file on the server side must be checked to see if it is open.

**Related Topics**

- OCILobClose()
  Closes a previously opened LOB or `BFILE`.
- OCILobFileCloseAll()
  Closes all open `BFILE`s on a given service context.
- OCILobFileExists()
  Tests to see if the `BFILE` exists on the server's operating system.
- OCILobFileClose()
  Closes a previously opened `BFILE`.
- OCILobFileIsOpen()
  Tests to see if the `BFILE` is open.
- OCILobFileOpen()
  Opens a `BFILE` on the file system of the server for read-only access.
- OCILobOpen()
  Opens a LOB, internal or external, in the indicated mode.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Functions for Opening and Closing LOBs
  OCI provides functions to explicitly open a LOB, `OCILobOpen()`, to close a LOB, `OCILobClose()`, and to test whether a LOB is open, `OCILobIsOpen()`.

## 26.3.28 OCILobIsTemporary()

Tests if a locator points to a temporary LOB.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 Topics

- OCILobCreateTemporary()
  Creates a temporary LOB.
- OCILobFreeTemporary()
  Frees a temporary LOB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.29 OCILobLoadFromFile2()

Loads and copies all or a portion of the file into an internal LOB. This function must be used for LOBs of size greater than 4 GB.

### Purpose

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

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**dst_locp (IN/OUT)**
A locator uniquely referencing the destination internal LOB, that 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 be in the same character set as the LOB in the database. No error checking is performed to verify this.

`OCILobLoadFromFile2()` throws an error when a remote locator is passed to it.

The source (`src_locp`) and the destination (`dst_locp`) LOBs must exist. If the data 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 `BLOB`s) or spaces (for `CLOB`s) 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) (see `OCILobLoadFromFile2()` to use for LOBs of size greater than 4 GB) 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 before 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 before performing this operation, then you must close it before you commit 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

### Related Topics

- OCILobAppend()
  Appends a LOB value at the end of another LOB as specified.
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobTrim2()
  Truncates the LOB value to a shorter length. This function must be used for LOBs of size greater than 4 GB.

- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.
- OCILobGetLength2()
  Gets the length of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.30 OCILobLocatorAssign()

Assigns one LOB or `BFILE` locator to another.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**src_locp (IN)**
The LOB or `BFILE` locator to copy from.

**dst_locpp (IN/OUT)**
The LOB or `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 `BFILE`s, only the locator that refers to the operating system file is copied to the table; the operating system file is not copied.

`OCILobLocatorAssign()` throws an error when a remote locator is passed to it.

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, but `OCILobAssign()` cannot be used for temporary LOBs.

> **Note:**
>
> If the `OCILobLocatorAssign()` function fails, the target locator is not 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 the source LOB locator is assigned to it.

If the source LOB locator refers to a temporary LOB, the destination is made into a temporary LOB too. The source and the destination are conceptually 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()` returns `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 point to the same LOB until any modification is made through either LOB locator. Hence `OCILobIsEqual()` returns `TRUE` right after `OCILobLocatorAssign()` until the first modification. In both these cases, after the `OCILobLocatorAssign()`, any changes to the source or the destination do 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 the two LOB locator pointers refer to the same LOB locator.

**Related Topics**

- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.
- OCILobAssign()
  Assigns one LOB or `BFILE` locator to another
- OCILobIsEqual()
  Compares two LOB or `BFILE` locators for equality.
- OCILobLocatorIsInit()
  Tests to see if a given LOB or `BFILE` locator is initialized.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.31 OCILobLocatorIsInit()

Tests to see if a given LOB or `BFILE` locator is initialized.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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; returns `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 these 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 Descriptor and LOB Locator Attributes

`BFILE` locators can be initialized by one of these methods:

- Selecting a non-`NULL` `BFILE` into the locator
- Pinning an object that contains a non-`NULL` `BFILE` attribute by `OCIObjectPin()`
- Calling OCILobFileSetName()

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.
- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCILobIsEqual()
  Compares two LOB or `BFILE` locators for equality.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.32 OCILobOpen()

Opens a LOB, internal or external, in the indicated mode.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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. `BFILE`s cannot be opened in read/write mode. If a user tries to write to a LOB or `BFILE` that was opened in read-only mode, an error is returned.

Opening a LOB requires a round-trip to the server for both internal and external LOBs. For internal LOBs, the open triggers other code that relies on the open call. For external LOBs (`BFILE`s), 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 to perform operations on it. When using function-based indexes, extensible indexes or context, and making multiple calls 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 ensures 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 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, 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

**Related Topics**

- OCILobClose()
  Closes a previously opened LOB or `BFILE`.
- OCILobFileCloseAll()
  Closes all open `BFILE`s on a given service context.
- OCILobFileExists()
  Tests to see if the `BFILE` exists on the server's operating system.
- OCILobFileClose()
  Closes a previously opened `BFILE`.
- OCILobFileIsOpen()
  Tests to see if the `BFILE` is open.
- OCILobFileOpen()
  Opens a `BFILE` on the file system of the server for read-only access.
- OCILobIsOpen()
  Tests whether a LOB or `BFILE` is open.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Functions for Opening and Closing LOBs
  OCI provides functions to explicitly open a LOB, `OCILobOpen()`, to close a LOB, `OCILobClose()`, and to test whether a LOB is open, `OCILobIsOpen()`.

## 26.3.33 OCILobRead2()

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.

**Purpose**

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,
                    void                *bufp,
                    oraub8               bufl,
ub1                  piece,
void                *ctxp,
OCICallbackLobRead2 (cbfp)
                    ( void           *ctxp,
                      const void     *bufp,
                      oraub8          lenp,
                      ub1             piecep
                      void          **changed_bufpp,
                      oraub8         *changed_lenp
                    )
ub2                  csid,
ub1                  csfrm );
```

### Parameters

**svchp (IN/OUT)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN)**
A LOB or `BFILE` locator that uniquely references the LOB or `BFILE`. This locator must have been 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 (`CLOB`s, `NCLOB`s), it is the number of characters from the beginning of the LOB; for binary LOBs or `BFILE`s, it is the number of bytes. The first position is 1.
If you use streaming (by polling or a callback), specify the offset in the first call; in subsequent polling calls, the offset parameter is ignored. When you use a callback, there is no offset parameter.

**bufp (IN/OUT)**
The pointer to a buffer into which the piece is read. The length of the allocated memory is assumed to be `bufl`.

**bufl (IN)**
The length of the buffer in octets. This value differs from the `amtp` value for `CLOB`s and for `NCLOB`s (`csfrm=SQLCS_NCHAR`) when the `amtp` parameter is specified in terms of characters, and 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 can be registered to be called for each piece. If this is `NULL`, then `OCI_NEED_DATA` is 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.
The `csfrm` parameter 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 you read or write LOBs, specify a character set form (`csfrm`) that matches the form of the locator itself.

For `BFILE`s, the operating system file must exist on the server, and it must have been opened by `OCILobFileOpen()` or `OCILobOpen()` using the input locator. Oracle Database must have permission to read the operating system file, and the user must have read permission on the directory object.

When you use the polling mode for `OCILobRead2()`, the first call must specify values for `offset` and `amtp`, but on subsequent polling calls to `OCILobRead2()`, you 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 apply to reading LOB data in streaming mode:

- When you use 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 `CLOB`s and `NCLOB`s, `char_amtp` gives the number of characters read in the current piece.

  For `CLOB`s and `NCLOB`s, if you do not pass `char_amtp`, then `char_amtp` is calculated internally as `byte_amtp/max char width`, so if `max char width` is 4, `char_amtp` is calculated as `byte_amtp`/4. Thus, `OCILobRead2()` does not calculate how many bytes are required for each character. Instead, `OCILobRead2()` fetches in the worst case the number of characters that can fit in `byte_amtp`. To fill the buffer, check the `byte_amtp` parameter to see how much of the buffer is filled, then call `OCILobRead2()` again to fetch the remaining bytes.
- When you use 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 cannot be filled with data.
- When you use polling, look at the `byte_amtp` parameter to see how much the buffer is filled for the current piece. For `CLOB`s and `NCLOB`s, `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.

**Related Topics**

- OCILobFileOpen()
  Opens a `BFILE` on the file system of the server for read-only access.
- OCILobOpen()
  Opens a LOB, internal or external, in the indicated mode.
- OCIBreak()
  Performs an immediate asynchronous break.
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobFileSetName()
  Sets the directory object and file name in the `BFILE` locator.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> - PL/SQL REF CURSORs and Nested Tables in OCI for additional information about Unicode format
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s
> - The demonstration programs included with your Oracle Database installation for a code sample showing the use of LOB reads and writes.
> - OCI Demonstration Programs
> - Runtime Data Allocation and Piecewise Operations in OCI for general information about piecewise OCI operations
> - Polling Mode Operations in OCI

## 26.3.34 OCILobSetContentType()

Sets a content type string for the data in the SecureFile to something that can be used by an application.

**Purpose**

Sets a content type string for the data in the SecureFile to something that can be used by an application.

**Syntax**

```
sword OCILobSetContentType ( OCIEnv        *envhp,
                             OCISvcCtx     *svchp,
                             OCIError      *errhp,
                             OCILobLocator *lobp,
                              const oratext *contenttypep,
                              ub4            contenttypelen,
                              ub4            mode);
```

### Parameters

**envhp (IN)**
The environment handle.

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that can be passed to `OCIErrorGet()` for diagnostic information when there is an error.

**lobp (IN)**
A LOB locator that uniquely references a LOB.

**contenttypep (IN)**
The `contenttype` to be set for the given LOB.

**contenttypelen (IN)**
The size of `contenttype` in bytes. The size must be less than or equal to `OCI_LOB_CONTENTTYPE_MAXSIZE` bytes.

**mode (IN)**
For future use. Pass zero now.

### Comments

This function only works on SecureFiles. If `lobp` is not a SecureFile, then the error `SECUREFILE_WRONGTYPE` is returned. If `lobp` is buffered, a temporary LOB, or an abstract LOB, then the error `SECUREFILE_BADLOB` is returned.

The maximum possible size of the `ContentType` string is defined as:

```
#define OCI_LOB_CONTENTTYPE_MAXSIZE 128
```

The `ContentType` is ASCII (that is, 1-byte/7-bit UTF8).

To clear an existing `contenttype` set on a SecureFile, invoke `OCILobSetContentType()` with `contenttypep` set to `(oratext *)0` and `contenttypelen` set to `0`.

### Related Topics

- OCILobGetContentType()
  Gets the user-specified content type string for the data in a SecureFile, if set.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.35 OCILobSetOptions()

Enables option settings for a SecureFile LOB.

### Purpose

Enables option settings for a SecureFile LOB.

### Syntax

```
sword OCILobSetOptions ( OCISvcCtx       *svchp,
                         OCIError        *errhp,
                         OCILobLocator   *locp,
                         ub4              option_types,
                         void            *optionsp,
                         ub4              optionslen,
                         ub4              mode );
```

### Parameters

**svchp (IN/OUT)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN/OUT)**
The LOB locator that uniquely references the LOB. This locator must have been a locator that was obtained from the server specified by `svchp`.

**option_types (IN)**
You can specify multiple option types and values by using the bit-wise inclusive OR ("|"). An error results if you specify an `option_types` value that is not enabled on the LOB column.

- Compression —`OCI_LOB_OPT_COMPRESS`
- Encryption — `OCI_LOB_OPT_ENCRYPT`
- Deduplication —`OCI_LOB_OPT_DEDUPLICATE`
- Allocation size — `OCI_LOB_OPT_ALLOCSIZE`
- Content type — `OCI_LOB_OPT_CONTENTTYPE`
- Modification time — `OCI_LOB_OPT_MODTIME`

**optionsp (IN)**
The possible settings are:

- `OCI_LOB_OPT_COMPRESS_OFF`
- `OCI_LOB_OPT_COMPRESS_ON`
- `OCI_LOB_ENCRYPT_OFF`
- `OCI_LOB_ENCRYPT_ON`

- `OCI_LOB_OPT_DEDUPLICATE_OFF`
- `OCI_LOB_OPT_DEDUPLICATE_ON`

**optionslen (IN)**
The length of the value in `optionsp`. Note that the only valid length at this time is `sizeof(ub4)`.

**mode (IN)**
Reserved for future use. Pass in 0.

**Related Topics**

- OCILobGetOptions()
  Obtains the enabled settings corresponding to the given input option types for a given SecureFile LOB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.36 OCILobTrim2()

Truncates the LOB value to a shorter length. This function must be used for LOBs of size greater than 4 GB.

**Purpose**

You can also use this function for LOBs smaller than 4 GB.

**Syntax**

```
sword OCILobTrim2 ( OCISvcCtx       *svchp,
                    OCIError        *errhp,
                    OCILobLocator   *locp,
                    oraub8           newlen );
```

**Parameters**

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN/OUT)**
An internal LOB locator that uniquely references the LOB. This locator must have been 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 before 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 before performing this operation, then you must close it before you commit 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

### Related Topics

- OCILobRead2()
  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.
- OCILobAppend()
  Appends a LOB value at the end of another LOB as specified.
- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.
- OCILobErase2()
  Erases a specified portion of the internal LOB data starting at a specified offset.
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.3.37 OCILobWrite2()

Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.

### Purpose

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,
void            *bufp,
oraub8           buflen,
ub1              piece,
void            *ctxp,
OCICallbackLobWrite2 (cbfp)
                (
                  void     *ctxp,
                  void     *bufp,
                  oraub8   *lenp,
                  ub1      *piecep
                  void    **changed_bufpp,
                  oraub8   *changed_lenp
                )
ub2              csid,
ub1              csfrm );
```

### Parameters

**svchp (IN/OUT)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN/OUT)**
An internal LOB locator that uniquely references the LOB. This locator must have been 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), specify the offset in the first call; in subsequent polling calls, the offset parameter is ignored. When you use a callback, there is no offset parameter.

**bufp (IN)**
The pointer to a buffer from which the piece is 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 results.

**buflen (IN)**
The length, in bytes, of the data in the buffer. This value differs from the `char_amtp` value for `CLOB`s and `NCLOB`s when the amount is specified in terms of characters using the `char_amtp` parameter, and 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 that the buffer is 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 can be registered to be called for each piece in a piecewise write. If this is `NULL`, the standard polling method is 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 the 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.
The `csfrm` parameter 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 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 you read or write LOBs, specify a character set form (`csfrm`) that matches the form of the locator itself.

When you use the polling mode for `OCILobWrite2()`, the first call must specify values for `offset`, `byte_amtp`, and `char_amtp`, but on subsequent polling calls to `OCILobWrite2()`, you 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 is invoked to get the next piece after a piece is written to the pipe. Each piece is 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, because 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 `CLOB`s and `NCLOB`s, `byte_amtp` and `char_amtp` return the data written by each piece in terms of number of bytes and number of characters respectively. For `BLOB`s, `byte_amtp` returns the number of bytes written by each piece, whereas `char_amtp` is undefined on output.

To write 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.

It is not mandatory that you wrap this LOB operation inside the open or close calls. If you did not open the LOB before 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 before performing this operation, then you must close it before you commit 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

**Related Topics**

- OCILobRead2()
  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.
- OCILobAppend()
  Appends a LOB value at the end of another LOB as specified.
- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> - PL/SQL REF CURSORs and Nested Tables in OCI for additional information about Unicode format
> - The demonstration programs included with your Oracle Database installation for a code sample showing the use of LOB reads and writes.
> - OCI Demonstration Programs
> - Runtime Data Allocation and Piecewise Operations in OCI for general information about piecewise OCI operations
> - Polling Mode Operations in OCI

## 26.3.38 OCILobWriteAppend2()

Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.

**Purpose**

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,
                           void                *bufp,
                           oraub8               buflen,
                           ub1                  piece,
                           void                *ctxp,
                           OCICallbackLobWrite2 (cbfp)
                                         (
                                           void     *ctxp,
                                           void     *bufp,
                                           oraub8   *lenp,
                                           ub1      *piecep
                                           void    **changed_bufpp,
                                           oraub8   *changed_lenp
                                          )
                           ub2                  csid,
                           ub1                  csfrm);
```

### Parameters

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 is 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 results.

**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 that the buffer is 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 can be registered to be called for each piece in a piecewise write. If this is `NULL`, the standard polling method is 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 the 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.
The `csfrm` parameter 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 is invoked to get the next piece after a piece is written to the pipe. Each piece is 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.

The `OCILobWriteAppend2()` function 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 `CLOB`s and `NCLOB`s, `byte_amtp` and `char_amtp` return the data written by each piece in terms of number of bytes and number of characters respectively. For `BLOB`s, `byte_amtp` returns the number of bytes written by each piece whereas `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 before 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 before performing this operation, then you must close it before you commit 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

**Related Topics**

- OCILobRead2()
  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.
- OCILobAppend()
  Appends a LOB value at the end of another LOB as specified.
- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> About Improving LOB Read/Write Performance

# 26.4 Database Advanced Queuing and Publish-Subscribe Functions

Lists and describes the Database Advanced Queuing and publish-subscribe functions.

Table 26-3 lists the Database Advanced Queuing and publish-subscribe functions that are described in this section. Use functions that end in "2" for all new applications.

> **See Also:**
>
> Table B-1 for Database Advanced Queuing programs

**Table 26-3 Advanced Queuing and Publish-Subscribe Functions**

| Function | Purpose |
|---|---|
| OCIAQDeq() | Performs an Advanced Queuing dequeue operation |
| OCIAQDeqArray() | Dequeue an array of messages |
| OCIAQEnq() | Performs an Advanced Queuing enqueue operation |
| OCIAQEnqArray() | Enqueue an array of messages |
| OCIAQListen2() | Listen on one or more queues on behalf of a list of agents. Supports buffered messaging and persistent queues. |
| OCISubscriptionDisable() | Disable a subscription registration to turn off notifications |
| OCISubscriptionEnable() | Enable notifications on a subscription |
| OCISubscriptionPost() | Post to a subscription to receive notifications |
| OCISubscriptionRegister() | Register a subscription |
| OCISubscriptionUnRegister() | Unregister a subscription |

## 26.4.1 OCIAQDeq()

### Purpose

Performs a dequeue operation using Database Advanced Queuing with OCI.

### Syntax

```
sword OCIAQDeq ( OCISvcCtx            *svch,
                 OCIError             *errh,
                 OraText              *queue_name,
                 OCIAQDeqOptions      *dequeue_options,
                 OCIAQMsgProperties   *message_properties,
                 OCIType              *payload_tdo,
                 void                **payload,
        void              **payload_ind,
        OCIRaw            **msgid,
        ub4                 flags );
```

### Parameters

**svch (IN)**
OCI service context.

**errh (IN)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 the following values:

- `OCI_MSG_PERSISTENT` (default)
- `OCI_MSG_BUFFERED`
- `OCI_MSG_PERSISTENT_OR_BUFFERED`

**message_properties (OUT)**
The message properties for the message; the properties are stored in an `OCIAQMsgProperties` descriptor, with OCI type constant `OCI_DTYPE_AQMSG_PROPERTIES`, which can have the following 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 that allow the user to set attributes of the payload, such as its text. For information about setting these attributes, see "Manipulating Object Attributes".

**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 to use this call. The OCI environment must be initialized in object mode (using `OCIEnvCreate()`, `OCIEnvNlsCreate()`), or `OCIInitialize()` (deprecated) to use this call.

### Related Topics

- OCIObjectFree()
  Frees and unpins an object instance.
- OCITypeByName()
  Gets the most current version of an existing TDO.
- OCITypeByRef()
  Gets a TDO when given a reference.
- OCIAQEnq()
  Performs an enqueue operation using Database Advanced Queuing.
- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> - OCI and Database Advanced Queuing
> - *Oracle Database Advanced Queuing User's Guide*

## 26.4.2 OCIAQDeqArray()

Dequeues an array of messages from a queue.

### Purpose

All messages in the array are dequeued with the same option and have 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,
                      void                **payload,
                      void                **payload_ind,
                      OCIRaw              **msgid,
                      void                 *ctxp,
                      OCICallbackAQDeq     (cbfp)
                                          (
                                          void          *ctxp,
                                          void         **payload,
                                          void         **payload_ind
                                          ),
                      ub4                   flags );
```

### Parameters

**svchp (IN)**
OCI service context (unchanged from `OCIAQDeq()`).

**errhp (IN)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 the following 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 the following 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 is passed to the callback function.

**cbfp (IN)**
The callback that can be registered to provide a buffer pointer into which the dequeued message is placed. If `NULL`, then messages are 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 to use this call. The OCI environment must be initialized in object mode (using `OCIEnvCreate()`, `OCIEnvNlsCreate()`), or `OCIInitialize()` (deprecated) 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 dequeuing, then the `OCIAQDeqArray()` function dequeues up to `iters` messages and returns immediately.

This function is not supported in nonblocking mode.

### Related Topics

- OCIAQDeq()
- OCIAQEnqArray()
  Enqueues an array of messages to a queue.
- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

**See Also:**

- OCI and Database Advanced Queuing
- *Oracle Database Advanced Queuing User's Guide*

## 26.4.3 OCIAQEnq()

Performs an enqueue operation using Database Advanced Queuing.

**Purpose**

Performs an enqueue operation using Database Advanced Queuing.

**Syntax**

```
sword OCIAQEnq ( OCISvcCtx            *svch,
                 OCIError             *errh,
                 OraText              *queue_name,
                 OCIAQEnqOptions      *enqueue_options,
                 OCIAQMsgProperties   *message_properties,
                 OCIType              *payload_tdo,
                 void                **payload,
                 void                **payload_ind,
                 OCIRaw              **msgid,
                 ub4                   flags );
```

**Parameters**

**svch (IN)**
OCI service context.

**errh (IN)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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. The properties are 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 the following 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`.
OCI provides functions that allow the user to set attributes of the payload, such as its text.

> **See Also:**
>
> Manipulating Object Attributes for information about setting these attributes

**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 to use this call.

The OCI environment must be initialized in object mode (using `OCIEnvCreate()`, `OCIEnvNlsCreate()`), or `OCIInitialize()` (deprecated) to use this call.

> **See Also:**
>
> - OCI and Database Advanced Queuing for more information about OCI and Advanced Queuing
> - *Oracle Database Advanced Queuing User's Guide*

To obtain a TDO for the payload, use `OCITypeByName()`, or `OCITypeByRef()`.

**Related Topics**

- OCITypeByName()
  Gets the most current version of an existing TDO.

- OCITypeByRef()
  Gets a TDO when given a reference.
- OCIAQDeq()
- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.4.4 OCIAQEnqArray()

Enqueues an array of messages to a queue.

### Purpose

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,
                      void                  **payload,
                      void                  **payload_ind,
                      OCIRaw                **msgid,
                      void                   *ctxp,
                      OCICallbackAQEnq       (cbfp)
                                            (
                                            void         *ctxp,
                                            void        **payload,
                                            void        **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 the following 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 is passed to the registered callback function.

**cbfp (IN)**
A callback that can be registered to provide messages dynamically. If `NULL`, then all messages must be materialized before calling `OCIAQEnqArray()`.

**flags (IN)**
Not currently used; pass as `OCI_DEFAULT`.

### Comments

This function is not supported in nonblocking mode.

### Related Topics

- OCIAQEnq()
  Performs an enqueue operation using Database Advanced Queuing.
- OCIAQDeqArray()
  Dequeues an array of messages from a queue.
- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.4.5 OCIAQListen2()

Listens on one or more queues on behalf of a list of agents.

**Purpose**

Supports buffered messaging and 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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_ATTR_WAIT` - Maximum wait time, in seconds, for the listen call
- `OCI_ATTR_MSG_DELIVERY_MODE` - Has one of these 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_ATTR_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 Topics

- OCIAQEnq()
  Performs an enqueue operation using Database Advanced Queuing.
- OCIAQDeq()
- OCISvcCtxToLda()
  Toggles between a V8 or later service context handle and a V7 `Lda_Def`.
- OCISubscriptionEnable()
  Enables a subscription registration that has been disabled.
- OCISubscriptionPost()
  Posts to a subscription that allows all clients who are registered for the subscription to get notifications.
- OCISubscriptionRegister()
  Registers a callback for message notification.
- OCISubscriptionUnRegister()
  Unregisters a subscription that turns off notifications.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.4.6 OCISubscriptionDisable()

Disables a subscription registration that turns off all notifications.

### Purpose

Disables a subscription registration that 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

**errhp (OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**mode (IN)**
Call-specific mode. The only valid value is `OCI_DEFAULT`. `OCI_DEFAULT` executes the default call that 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 after an `OCISubscriptionDisable()` are discarded by the system until an `OCISubscriptionEnable()` is performed.

### Related Topics

- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.
- OCISubscriptionEnable()
  Enables a subscription registration that has been disabled.
- OCISubscriptionPost()
  Posts to a subscription that allows all clients who are registered for the subscription to get notifications.
- OCISubscriptionRegister()
  Registers a callback for message notification.
- OCISubscriptionUnRegister()
  Unregisters a subscription that turns off notifications.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.4.7 OCISubscriptionEnable()

Enables a subscription registration that has been disabled.

### Purpose

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:**
>
> Subscription Handle Attributes

**errhp (OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**mode (IN)**
Call-specific mode. The only valid value is OCI_DEFAULT. This value executes the default call that 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 Topics

- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.
- OCISvcCtxToLda()
  Toggles between a V8 or later service context handle and a V7 `Lda_Def`.
- OCISubscriptionRegister()
  Registers a callback for message notification.
- OCISubscriptionUnRegister()
  Unregisters a subscription that turns off notifications.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.4.8 OCISubscriptionPost()

Posts to a subscription that allows all clients who are registered for the subscription to get notifications.

### Purpose

Posts to a subscription that 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:**
>
> Subscription Handle Attributes

The `OCI_ATTR_SUBSCR_PAYLOAD` attribute must be set for each subscription handle before 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 must 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**mode (IN)**
Call-specific mode. The only valid value is OCI_DEFAULT. This value 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 for several system events. If the application needs more rigid guarantees, it can use the Advanced Queuing functionality by enqueuing to the queue.

### Related Topics

- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.
- OCISvcCtxToLda()
  Toggles between a V8 or later service context handle and a V7 `Lda_Def`.
- OCISubscriptionEnable()
  Enables a subscription registration that has been disabled.
- OCISubscriptionRegister()
  Registers a callback for message notification.
- OCISubscriptionUnRegister()
  Unregisters a subscription that turns off notifications.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.4.9 OCISubscriptionRegister()

Registers a callback for message notification.

### 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 is returned.

One of the following attributes must also be set:

- `OCI_ATTR_SUBSCR_CALLBACK`
- `OCI_ATTR_SUBSCR_CTX`
- `OCI_ATTR_SUBSCR_RECPT`

> **See Also:**
>
> Subscription Handle Attributes for information about the handle attributes

When a notification is received for the registration denoted by `subscrhpp[i]`, either the user-defined callback function (`OCI_ATTR_SUBSCR_CALLBACK`) set for `subscrhpp[i]` is invoked with the context (`OCI_ATTR_SUBSCR_CTX`) set for `subscrhpp[i]`, or an email is sent to (`OCI_ATTR_SUBSCR_RECPT`) set for `subscrhpp[i]`, or the PL/SQL procedure (`OCI_ATTR_SUBSCR_RECPT`) set for `subscrhpp[i]` is 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**mode (IN)**
Call-specific mode. Valid values are:

- `OCI_DEFAULT` - Use when there is only one server DN in the server DN descriptor. The registration request is sent to the database. If a database connection is not available, the registration request is detoured to the LDAP server.
- `OCI_REG_LDAPONLY` - The registration request is sent directly to the LDAP server. Use this mode when there are multiple server DNs in the server DN descriptor, or you are certain that a database connection is not available.

Whenever a new client process comes up, or an old one goes down and comes back up, it must register for all subscriptions of interest. If the client stays up and the server first goes down and then comes back up, the client continues to receive notifications for registrations that are DISCONNECTED. However, the client does not receive notifications for CONNECTED registrations, as they are lost after the server goes down and comes back up.

### Comments

This call is invoked for registration to a subscription that identifies the subscription name of interest and the associated callback to be invoked. Interest in several subscriptions can be registered simultaneously.

This interface is only valid for the asynchronous mode of message delivery. In this mode, a subscriber issues a registration call that 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 multiconsumer queue. The string should be in uppercase.

Each namespace has its own privilege model. If the user performing the registration is not entitled to register in the namespace for the specified subscription, an error is returned.

**Related Topics**

- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.
- OCISvcCtxToLda()
  Toggles between a V8 or later service context handle and a V7 `Lda_Def`.
- OCISubscriptionEnable()
  Enables a subscription registration that has been disabled.
- OCISubscriptionPost()
  Posts to a subscription that allows all clients who are registered for the subscription to get notifications.
- OCISubscriptionUnRegister()
  Unregisters a subscription that turns off notifications.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.4.10 OCISubscriptionUnRegister()

Unregisters a subscription that turns off notifications.

**Purpose**

Unregisters a subscription that 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:**
>
> Subscription Handle Attributes

**errhp (OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**mode (IN)**
Call-specific mode. Valid values are:

- `OCI_DEFAULT` - Use when there is only one server DN in the server DN descriptor. The registration request is sent to the database. If a database connection is not available, the registration request is detoured to the LDAP server.
- `OCI_REG_LDAPONLY` - The registration request is sent directly to the LDAP server. Use this mode when there are multiple server DNs in the server DN descriptor, or you are certain that a database connection is not available.

### Comments

Unregistering a subscription ensures that the user does not receive notifications regarding the specified subscription in the future. If the user wants to resume notification, then the only option is to reregister for the subscription.

All notifications that would otherwise have been delivered are not delivered after a subsequent registration is performed because the user is no longer in the list of interested clients.

### Related Topics

- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.
- OCISvcCtxToLda()
  Toggles between a V8 or later service context handle and a V7 `Lda_Def`.
- OCISubscriptionEnable()
  Enables a subscription registration that has been disabled.
- OCISubscriptionPost()
  Posts to a subscription that allows all clients who are registered for the subscription to get notifications.
- OCISubscriptionRegister()
  Registers a callback for message notification.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 26.5 Direct Path Loading Functions

Lists and describes the direct path loading functions.

Table 26-4 lists the direct path loading functions that are described in this section.

**Table 26-4    Direct Path Loading Functions**

| Function | Purpose |
|---|---|
| OCIDirPathAbort() | Terminate a direct path operation |
| OCIDirPathColArrayEntryGet() | Get a specified entry in a column array |
| OCIDirPathColArrayEntrySet() | Set a specified entry in a column array to a specific value |
| OCIDirPathColArrayReset() | Reset the row array state |
| OCIDirPathColArrayRowGet() | Get the base row pointers for a specified row number |
| OCIDirPathColArrayToStream() | Convert from a column array to a direct path stream format |
| OCIDirPathDataSave() | Do a data savepoint, or commit the loaded data and finish the load operation |
| OCIDirPathFinish() | Finish and commit the loaded data |
| OCIDirPathFlushRow() | Deprecated. |
| OCIDirPathLoadStream() | Load the data converted to direct path stream format |
| OCIDirPathPrepare() | Prepare direct path interface to convert or load rows |
| OCIDirPathStreamReset() | Reset the direct path stream state |

## 26.5.1 OCIDirPathAbort()

Terminates a direct path operation.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

### Comments

All state that was maintained by the server on behalf of the direct path operation is destroyed by a termination. For a direct path load, the data loaded before the terminate operation is not 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 Topics

- OCIDirPathFinish()
  Finishes the direct path load operation.
- OCIDirPathPrepare()
- OCIDirPathLoadStream()
  Loads the data converted to direct path stream format.
- OCIDirPathStreamReset()
  Resets the direct path stream state.
- OCIDirPathDataSave()
  Depending on the action requested, does a data savepoint, or commits the loaded data and finishes the direct path load operation.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.2 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**rownum (IN)**
Zero-based row offset.

**colIdx (IN)**
The column's index used when building the column parameter list.

**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 these values is returned:

- `OCI_DIRPATH_COL_COMPLETE` - All data for the 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 `cvalpp` and `clenp` parameters are not set by this operation.

### Related Topics

- OCIDirPathColArrayEntrySet()
  Sets a specified entry in a column array to the supplied values.
- OCIDirPathColArrayRowGet()
  Gets the column array row pointers for a given row number.
- OCIDirPathColArrayReset()
  Resets the column array state.
- OCIDirPathColArrayToStream()
  Converts from column array format to a direct path stream format.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.3 OCIDirPathColArrayEntrySet()

Sets a specified entry in a column array to the supplied values.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**rownum (IN)**
Zero-based row offset.

**colIdx (IN)**
The column's index used when building the column parameter list.

**cvalp (IN)**
Pointer to column data.

**clen (IN)**
Length of column data.

**cflg (IN)**
Column flag. One of these values is returned:

- `OCI_DIRPATH_COL_COMPLETE` - All data for the 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 `cvalp` 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 Topics

- OCIDirPathColArrayEntryGet()
- OCIDirPathColArrayRowGet()
  Gets the column array row pointers for a given row number.
- OCIDirPathColArrayReset()
  Resets the column array state.
- OCIDirPathColArrayToStream()
  Converts from column array format to a direct path stream format.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.4 OCIDirPathColArrayReset()

Resets the column array state.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 Topics

- OCIDirPathColArrayEntryGet()
- OCIDirPathColArrayEntrySet()
  Sets a specified entry in a column array to the supplied values.
- OCIDirPathColArrayRowGet()
  Gets the column array row pointers for a given row number.
- OCIDirPathColArrayToStream()
  Converts from column array format to a direct path stream format.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.5 OCIDirPathColArrayRowGet()

Gets the column array row pointers for a given row number.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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. You can use this interface 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 dereferencing memory beyond the column array boundaries. The dimensions of the column array are available as attributes of the column array.

**Related Topics**

- OCIDirPathColArrayEntryGet()
- OCIDirPathColArrayEntrySet()
  Sets a specified entry in a column array to the supplied values.
- OCIDirPathColArrayReset()
  Resets the column array state.
- OCIDirPathColArrayToStream()
  Converts from column array format to a direct path stream format.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.6 OCIDirPathColArrayToStream()

Converts from column array format to a direct path stream format.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 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 Database 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 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 that 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 `OCIDirPathColArrayToStream()` again to convert the remainder of the column array data. The column array attribute `OCI_ATTR_ROW_COUNT` is the number of rows successfully converted in the last call. The row offset must be updated for the next conversion; internal state does keep track of the column to continue conversion from. The `OCI_ATTR_ROW_COUNT` value must be added to the previous row offset by the caller.
- `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 that is marked partial.

**Related Topics**

- OCIDirPathLoadStream()
  Loads the data converted to direct path stream format.
- OCIDirPathColArrayEntryGet()
- OCIDirPathColArrayEntrySet()
  Sets a specified entry in a column array to the supplied values.
- OCIDirPathColArrayReset()
  Resets the column array state.
- OCIDirPathColArrayRowGet()
  Gets the column array row pointers for a given row number.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.7 OCIDirPathDataSave()

Depending on the action requested, does a data savepoint, or commits the loaded data and finishes the direct path load operation.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**action (IN)**
Values for action parameter to `OCIDirPathDataSave()` are as follows:

- `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 backend 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 different from properly terminating the load, because resources allocated are not freed.

**Related Topics**

- OCIDirPathAbort()
  Terminates a direct path operation.
- OCIDirPathFinish()
  Finishes the direct path load operation.
- OCIDirPathPrepare()
- OCIDirPathStreamReset()
  Resets the direct path stream state.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.8 OCIDirPathFinish()

Finishes the direct path load operation.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 backend has properly terminated the load.

### Related Topics

- OCIDirPathAbort()
  Terminates a direct path operation.
- OCIDirPathDataSave()
  Depending on the action requested, does a data savepoint, or commits the loaded data and finishes the direct path load operation.
- OCIDirPathPrepare()
- OCIDirPathStreamReset()
  Resets the direct path stream state.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.9 OCIDirPathFlushRow()

Flushes a partially loaded row from the server. This function is deprecated.

### Purpose

Flushes a partially loaded row from the 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 does nothing.

### Related Topics

- OCIDirPathAbort()
  Terminates a direct path operation.

- OCIDirPathFinish()
  Finishes the direct path load operation.
- OCIDirPathPrepare()
- OCIDirPathLoadStream()
  Loads the data converted to direct path stream format.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.10 OCIDirPathLoadStream()

Loads the data converted to direct path stream format.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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, a 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 must 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 that 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 Topics**

- OCIDirPathAbort()
  Terminates a direct path operation.
- OCIDirPathDataSave()
  Depending on the action requested, does a data savepoint, or commits the loaded data and finishes the direct path load operation.
- OCIDirPathFinish()
  Finishes the direct path load operation.
- OCIDirPathPrepare()
- OCIDirPathStreamReset()
  Resets the direct path stream state.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.11 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

### Comments

After the name of the object to be operated on is set, the external attributes of the column data are 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 backend 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 Topics**

- OCIDirPathAbort()
  Terminates a direct path operation.
- OCIDirPathDataSave()
  Depending on the action requested, does a data savepoint, or commits the loaded data and finishes the direct path load operation.
- OCIDirPathFinish()
  Finishes the direct path load operation.
- OCIDirPathStreamReset()
  Resets the direct path stream state.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.5.12 OCIDirPathStreamReset()

Resets the direct path stream state.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 Topics**

- OCIDirPathAbort()
  Terminates a direct path operation.

- OCIDirPathDataSave()
  Depending on the action requested, does a data savepoint, or commits the loaded data and finishes the direct path load operation.
- OCIDirPathFinish()
  Finishes the direct path load operation.
- OCIDirPathPrepare()
- OCIDirPathColArrayToStream()
  Converts from column array format to a direct path stream format.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6 Thread Management Functions

Lists and describes the thread management functions.

Table 26-5 lists the thread management functions that are described in this section.

**Table 26-5 Thread Management Functions**

| Function | Purpose |
|---|---|
| OCIThreadClose() | Close a thread handle |
| OCIThreadCreate() | Create a new thread |
| OCIThreadHandleGet() | Retrieve the OCIThreadHandle of the thread in which it is called |
| OCIThreadHndDestroy() | Destroy and deallocate the thread handle |
| OCIThreadHndInit() | Allocate and initialize the thread handle |
| OCIThreadIdDestroy() | Destroy and deallocate a thread ID |
| OCIThreadIdGet() | Retrieve the `OCIThreadId` of the thread in which it is called |
| OCIThreadIdInit() | Allocate and initialize the thread ID |
| OCIThreadIdNull() | Determine whether a given `OCIThreadId` is the `NULL` thread ID |
| OCIThreadIdSame() | Determine whether two `OCIThreadIds` represent the same thread |
| OCIThreadIdSet() | Set one `OCIThreadId` to another |
| OCIThreadIdSetNull() | Set the `NULL` thread ID to a given `OCIThreadId` |
| OCIThreadInit() | Initialize OCIThread context |
| OCIThreadIsMulti() | Tell the caller whether the application is running in a multithreaded environment or a single-threaded environment |
| OCIThreadJoin() | Allow the calling thread to join with another thread |
| OCIThreadKeyDestroy() | Destroy and deallocate the key pointed to by `key` |
| OCIThreadKeyGet() | Get the calling thread's current value for a key |

**Table 26-5 (Cont.) Thread Management Functions**

| Function | Purpose |
|---|---|
| OCIThreadKeyInit() | Create a key |
| OCIThreadKeySet() | Set the calling thread's value for a key |
| OCIThreadMutexAcquire() | Acquire a mutex for the thread in which it is called |
| OCIThreadMutexDestroy() | Destroy and deallocate a mutex |
| OCIThreadMutexInit() | Allocate and initialize a mutex |
| OCIThreadMutexRelease() | Release a mutex |
| OCIThreadProcessInit() | Perform OCIThread process initialization |
| OCIThreadTerm() | Release the OCIThread context |

## 26.6.1 OCIThreadClose()

Closes a thread handle.

### Purpose

Closes a thread handle.

### Syntax

```
sword OCIThreadClose ( void             *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

The `tHnd` parameter should be initialized by `OCIThreadHndInit()`. Both the thread handle and the thread ID that was returned by the same call to `OCIThreadCreate()` are invalid after the call to `OCIThreadClose()`.

### Related Topics

- OCIThreadCreate()
  Creates a new thread.
- OCIThreadHndInit()
  Allocates and initializes the thread handle.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.2 OCIThreadCreate()

Creates a new thread.

### Purpose

Creates a new thread.

### Syntax

```
sword OCIThreadCreate ( void            *hndl,
                        OCIError        *err,
                        void (*start)   (void *),
                        void            *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 terminates. The function should not return a value and should accept one parameter, a `void`. 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` is not valid in the calling thread because the timing of the spawned threads termination is unknown.

The `tid` parameter should be initialized by `OCIThreadIdInit()` and `tHnd` should be initialized by `OCIThreadHndInit()`.

### Related Topics

- OCIThreadClose()
  Closes a thread handle.
- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadHndInit()
  Allocates and initializes the thread handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.3 OCIThreadHandleGet()

Retrieves the `OCIThreadHandle` of the thread in which it is called.

### Purpose

Retrieves the `OCIThreadHandle` of the thread in which it is called.

### Syntax

```
sword OCIThreadHandleGet ( void            *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

The `tHnd` parameter 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 Topics

- OCIThreadHndDestroy()
  Destroys and deallocates the thread handle.
- OCIThreadHndInit()
  Allocates and initializes the thread handle.
- OCIThreadClose()
  Closes a thread handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.4 OCIThreadHndDestroy()

Destroys and deallocates the thread handle.

### Purpose

Destroys and deallocates the thread handle.

### Syntax

```
sword OCIThreadHndDestroy ( void              *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

The `thnd` parameter should be initialized by `OCIThreadHndInit()`.

**Related Topics**

- OCIThreadHandleGet()
  Retrieves the `OCIThreadHandle` of the thread in which it is called.
- OCIThreadHndInit()
  Allocates and initializes the thread handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.5 OCIThreadHndInit()

Allocates and initializes the thread handle.

### Purpose

Allocates and initializes the thread handle.

### Syntax

```
sword OCIThreadHndInit ( void              *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 the pointer to the thread handle to initialize.

### Related Topics

- OCIThreadHandleGet()
  Retrieves the `OCIThreadHandle` of the thread in which it is called.
- OCIThreadHndDestroy()
  Destroys and deallocates the thread handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.6 OCIThreadIdDestroy()

Destroys and deallocates a thread ID.

### Purpose

Destroys and deallocates a thread ID.

### Syntax

```
sword OCIThreadIdDestroy (void           *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

The `tid` parameter should be initialized by `OCIThreadHndInit()`.

### Related Topics

- OCIThreadIdGet()
  Retrieves the `OCIThreadId` of the thread in which it is called.

- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadIdNull()
  Determines whether a given `OCIThreadId` is the `NULL` thread ID.
- OCIThreadIdSame()
  Determines whether two `OCIThreadIds` represent the same thread.
- OCIThreadIdSet()
  Sets one `OCIThreadId` to another.
- OCIThreadIdSetNull()
  Sets the `NULL` thread ID to a given `OCIThreadId`.
- OCIThreadHndInit()
  Allocates and initializes the thread handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.7 OCIThreadIdGet()

Retrieves the `OCIThreadId` of the thread in which it is called.

### Purpose

Retrieves the `OCIThreadId` of the thread in which it is called.

### Syntax

```
sword OCIThreadIdGet ( void         *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

The `tid` parameter should be initialized by `OCIThreadHndInit()`. When OCIThread is used in a single-threaded environment, `OCIThreadIdGet()` always places the same value in the location pointed to by `tid`. The exact value itself is not important. The important thing is that it is different from the `NULL` thread ID and that it is always the same value.

### Related Topics

- OCIThreadIdDestroy()
  Destroys and deallocates a thread ID.

- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadIdNull()
  Determines whether a given `OCIThreadId` is the `NULL` thread ID.
- OCIThreadIdSame()
  Determines whether two `OCIThreadId`s represent the same thread.
- OCIThreadIdSet()
  Sets one `OCIThreadId` to another.
- OCIThreadIdSetNull()
  Sets the `NULL` thread ID to a given `OCIThreadId`.
- OCIThreadHndInit()
  Allocates and initializes the thread handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.8 OCIThreadIdInit()

Allocate and initialize the thread ID `tid`.

### Purpose

Allocate and initialize the thread ID `tid`.

### Syntax

```
sword OCIThreadIdInit ( void          *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 Topics

- OCIThreadIdDestroy()
  Destroys and deallocates a thread ID.
- OCIThreadIdGet()
  Retrieves the `OCIThreadId` of the thread in which it is called.
- OCIThreadIdNull()
  Determines whether a given `OCIThreadId` is the `NULL` thread ID.
- OCIThreadIdSame()
  Determines whether two `OCIThreadId`s represent the same thread.

- OCIThreadIdSet()
  Sets one `OCIThreadId` to another.
- OCIThreadIdSetNull()
  Sets the `NULL` thread ID to a given `OCIThreadId`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

**Related Topics**

- OCIThreadTerm()
  Releases the OCIThread context.
- OCIThreadProcessInit()
  Performs OCIThread process initialization.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.9 OCIThreadIdNull()

Determines whether a given `OCIThreadId` is the `NULL` thread ID.

### Purpose

Determines whether a given `OCIThreadId` is the `NULL` thread ID.

### Syntax

```
sword OCIThreadIdNull ( void         *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`. The `tid` parameter should be initialized by `OCIThreadIdInit()`.

**Related Topics**

- OCIThreadIdDestroy()
  Destroys and deallocates a thread ID.

- OCIThreadIdGet()
  Retrieves the `OCIThreadId` of the thread in which it is called.
- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadIdSame()
  Determines whether two `OCIThreadIds` represent the same thread.
- OCIThreadIdSet()
  Sets one `OCIThreadId` to another.
- OCIThreadIdSetNull()
  Sets the `NULL` thread ID to a given `OCIThreadId`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.10 OCIThreadIdSame()

Determines whether two `OCIThreadIds` represent the same thread.

### Purpose

Determines whether two `OCIThreadIds` represent the same thread.

### Syntax

```
sword OCIThreadIdSame ( void          *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`. The `result` parameter is set to `TRUE` if both `tid1` and `tid2` are the `NULL` thread ID. The parameters `tid1` and `tid2` should be initialized by `OCIThreadIdInit()`.

**Related Topics**

- OCIThreadIdDestroy()
  Destroys and deallocates a thread ID.
- OCIThreadIdGet()
  Retrieves the `OCIThreadId` of the thread in which it is called.
- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadIdNull()
  Determines whether a given `OCIThreadId` is the `NULL` thread ID.
- OCIThreadIdSet()
  Sets one `OCIThreadId` to another.
- OCIThreadIdSetNull()
  Sets the `NULL` thread ID to a given `OCIThreadId`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.11 OCIThreadIdSet()

Sets one `OCIThreadId` to another.

### Purpose

Sets one `OCIThreadId` to another.

### Syntax

```
sword OCIThreadIdSet ( void          *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 set to.

**tidSrc (IN)**
This should point to the `OCIThreadId` to set from.

### Comments

The `tid` parameter should be initialized by `OCIThreadIdInit()`.

**Related Topics**

- OCIThreadIdDestroy()
  Destroys and deallocates a thread ID.

- OCIThreadIdGet()
  Retrieves the `OCIThreadId` of the thread in which it is called.
- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadIdNull()
  Determines whether a given `OCIThreadId` is the `NULL` thread ID.
- OCIThreadIdSame()
  Determines whether two `OCIThreadIds` represent the same thread.
- OCIThreadIdSetNull()
  Sets the `NULL` thread ID to a given `OCIThreadId`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.12 OCIThreadIdSetNull()

Sets the `NULL` thread ID to a given `OCIThreadId`.

### Purpose

Sets the `NULL` thread ID to a given `OCIThreadId`.

### Syntax

```
sword OCIThreadIdSetNull ( void          *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

The `tid` parameter should be initialized by `OCIThreadIdInit()`.

### Related Topics

- OCIThreadIdDestroy()
  Destroys and deallocates a thread ID.
- OCIThreadIdGet()
  Retrieves the `OCIThreadId` of the thread in which it is called.
- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadIdNull()
  Determines whether a given `OCIThreadId` is the `NULL` thread ID.

- OCIThreadIdSame()
  Determines whether two `OCIThreadIds` represent the same thread.
- OCIThreadIdSet()
  Sets one `OCIThreadId` to another.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.13 OCIThreadInit()

Initializes the OCIThread context.

### Purpose

Initializes the OCIThread context.

### Syntax

```
sword OCIThreadInit ( void      *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 to 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 OCIThread context. It also saves a pointer to the context in some system-dependent manner. Subsequent calls to `OCIThreadInit()` return the same context.

Each call to `OCIThreadInit()` must eventually be matched by a call to `OCIThreadTerm()`.

## 26.6.14 OCIThreadIsMulti()

Tells the caller whether the application is running in a multithreaded environment or a single-threaded environment.

### 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 Topics**

- OCIThreadIdDestroy()
  Destroys and deallocates a thread ID.
- OCIThreadIdGet()
  Retrieves the `OCIThreadId` of the thread in which it is called.
- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadIdNull()
  Determines whether a given `OCIThreadId` is the `NULL` thread ID.
- OCIThreadIdSame()
  Determines whether two `OCIThreadId`s represent the same thread.
- OCIThreadIdSet()
  Sets one `OCIThreadId` to another.

## 26.6.15 OCIThreadJoin()

Allows the calling thread to join with another thread.

### Purpose

Allows the calling thread to join with another thread.

### Syntax

```
sword OCIThreadJoin ( void            *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.

The `tHnd` parameter should be initialized by `OCIThreadHndInit()`. The result of multiple threads all trying to join with the same thread is undefined.

**Related Topics**

- OCIThreadIdDestroy()
  Destroys and deallocates a thread ID.
- OCIThreadIdGet()
  Retrieves the `OCIThreadId` of the thread in which it is called.
- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadIdNull()
  Determines whether a given `OCIThreadId` is the `NULL` thread ID.
- OCIThreadIdSame()
  Determines whether two `OCIThreadId`s represent the same thread.
- OCIThreadIdSet()
  Sets one `OCIThreadId` to another.
- OCIThreadHndInit()
  Allocates and initializes the thread handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.16 OCIThreadKeyDestroy()

Destroys and deallocates the key pointed to by `key`.

**Purpose**

Destroys and deallocates the key pointed to by `key`.

**Syntax**

```
sword OCIThreadKeyDestroy ( void            *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. The function `OCIThreadKeyDestroy()` is used to terminate any resources that the OCIThread acquired when it created `key`. The `OCIThreadKeyDestFunc` callback of `OCIThreadKeyInit()` is a key value destructor; it does not operate on the key itself.

This must be called after the user has finished using the key. Not calling the `OCIThreadKeyDestroy()` function may result in memory leaks.

**Related Topics**

- OCIThreadKeyGet()
  Gets the calling thread's current value for a key.
- OCIThreadKeyInit()
  Creates a key.
- OCIThreadKeySet()
  Sets the calling thread's value for a key.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.17 OCIThreadKeyGet()

Gets the calling thread's current value for a key.

### Purpose

Gets the calling thread's current value for a key.

### Syntax

```
sword OCIThreadKeyGet ( void            *hndl,
                        OCIError        *err,
                        OCIThreadKey    *key,
                        void           **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 Topics**

- OCIThreadKeyDestroy()
  Destroys and deallocates the key pointed to by `key`.

- OCIThreadKeyInit()
  Creates a key.
- OCIThreadKeySet()
  Sets the calling thread's value for a key.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.18 OCIThreadKeyInit()

Creates a key.

### Purpose

Creates a key.

### Syntax

```
sword OCIThreadKeyInit (void                   *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 allocates 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 returned. That key can be used with `OCIThreadKeyGet()` and `OCIThreadKeySet()`. The initial value of the key is `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` is called whenever a thread that has a non-`NULL` value for the key terminates. The routine is called with one parameter. The parameter is the key's 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 Topics

- OCIThreadKeyDestroy()
  Destroys and deallocates the key pointed to by `key`.

- OCIThreadKeyGet()
  Gets the calling thread's current value for a key.
- OCIThreadKeySet()
  Sets the calling thread's value for a key.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.19 OCIThreadKeySet()

Sets the calling thread's value for a key.

### Purpose

Sets the calling thread's value for a key.

### Syntax

```
sword OCIThreadKeySet ( void           *hndl,
                        OCIError       *err,
                        OCIThreadKey   *key,
                        void           *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 Topics

- OCIThreadKeyDestroy()
  Destroys and deallocates the key pointed to by `key`.
- OCIThreadKeyGet()
  Gets the calling thread's current value for a key.
- OCIThreadKeyInit()
  Creates a key.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.20 OCIThreadMutexAcquire()

Acquires a mutex for the thread in which it is called.

### Purpose

Acquires a mutex for the thread in which it is called.

### Syntax

```
sword OCIThreadMutexAcquire ( void            *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 function's behavior is undefined if it is used by a thread to acquire a mutex that is already held by that thread.

### Related Topics

- OCIThreadMutexDestroy()
  Destroys and deallocates a mutex.
- OCIThreadMutexInit()
  Allocates and initializes a mutex.
- OCIThreadMutexRelease()
  Releases a mutex.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.21 OCIThreadMutexDestroy()

Destroys and deallocates a mutex.

### Purpose

Destroys and deallocates a mutex.

### Syntax

```
sword OCIThreadMutexDestroy ( void            *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 after 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 Topics

- OCIThreadMutexAcquire()
  Acquires a mutex for the thread in which it is called.
- OCIThreadMutexInit()
  Allocates and initializes a mutex.
- OCIThreadMutexRelease()
  Releases a mutex.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.22 OCIThreadMutexInit()

Allocates and initializes a mutex.

### Purpose

Allocates and initializes a mutex.

### Syntax

```
sword OCIThreadMutexInit ( void             *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 before 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 Topics

- OCIThreadMutexDestroy()
  Destroys and deallocates a mutex.
- OCIThreadMutexAcquire()
  Acquires a mutex for the thread in which it is called.
- OCIThreadMutexRelease()
  Releases a mutex.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.23 OCIThreadMutexRelease()

Releases a mutex.

### Purpose

Releases a mutex.

### Syntax

```
sword OCIThreadMutexRelease ( void            *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 acquires it and becomes 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 Topics**

- OCIThreadMutexDestroy()
  Destroys and deallocates a mutex.
- OCIThreadMutexInit()
  Allocates and initializes a mutex.
- OCIThreadMutexAcquire()
  Acquires a mutex for the thread in which it is called.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.6.24 OCIThreadProcessInit()

Performs OCIThread process initialization.

**Purpose**

Performs OCIThread process initialization.

**Syntax**

```
void OCIThreadProcessInit ( );
```

**Comments**

Whether this function must be called depends on how OCIThread 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 do 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 do not have any effect.

**Related Topics**

- OCIThreadIdDestroy()
  Destroys and deallocates a thread ID.
- OCIThreadIdGet()
  Retrieves the `OCIThreadId` of the thread in which it is called.
- OCIThreadIdInit()
  Allocate and initialize the thread ID `tid`.
- OCIThreadIdNull()
  Determines whether a given `OCIThreadId` is the `NULL` thread ID.
- OCIThreadIdSame()
  Determines whether two `OCIThreadIds` represent the same thread.

- OCIThreadIdSet()
  Sets one `OCIThreadId` to another.

## 26.6.25 OCIThreadTerm()

Releases the OCIThread context.

**Purpose**

Releases the OCIThread context.

**Syntax**

```
sword OCIThreadTerm ( void      *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()` does not do anything until it has been called as many times as `OCIThreadInit()` has been called. When that happens, `OCIThreadTerm()` terminates the OCIThread layer and frees the memory allocated for the context. Once this happens, the context should not be reused. It is necessary to obtain a new one by calling `OCIThreadInit()`.

**Related Topics**

- OCIThreadInit()
  Initializes the OCIThread context.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 26.7 Transaction Functions

Lists and describes the transaction functions.

Table 26-6 lists the transaction functions that are described in this section.

**Table 26-6 Transaction Functions**

| Function | Purpose |
|---|---|
| OCITransCommit() | Commit a transaction on a service context |
| OCITransDetach() | Detach a transaction from a service context |
| OCITransForget() | Forget a prepared global transaction |

**Table 26-6 (Cont.) Transaction Functions**

| Function | Purpose |
|---|---|
| OCITransMultiPrepare() | Prepare a transaction with multiple branches in a single cell |
| OCITransPrepare() | Prepare a global transaction for commit |
| OCITransRollback() | Roll back a transaction |
| OCITransStart() | Start a transaction on a service context |

## 26.7.1 OCITransCommit()

Commits the transaction associated with a specified service context.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**flags (IN)**
A flag used for one-phase commit optimization in global transactions.
`OCI_DEFAULT` - If the transaction is nondistributed, 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 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.

> **Note:**
>
> There is a potential for silent transaction loss when you use `OCI_TRANS_WRITENOWAIT`. Transaction loss occurs silently with shutdown abort, startup force, and any instance or node failure. On an Oracle RAC system, asynchronously committed changes may not be immediately available to read on other instances.

These last four options only affect the commit of top-level nondistributed 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 code example demonstrates the use of a simple local transaction, as described in "Simple Local Transactions".

Using OCITransCommit() in a Simple Local Transaction

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  OCIStmt *stmthp;
  void      *tmp;
  text sqlstmt[128];
  OCIEnvCreate(&envhp, OCI_DEFAULT, (void  *)0, 0, 0, 0,
        (size_t)0, (void  *)0);

  OCIHandleAlloc( (void  *) envhp, (void  **) &errhp, (ub4) OCI_HTYPE_ERROR,
                (size_t)0, (void  **) 0);
  OCIHandleAlloc( (void  *) envhp, (void  **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                (size_t)0, (void  **) 0);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (void  *) envhp, (void  **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                (size_t)0, (void  **) 0);

  OCIHandleAlloc((void  *)envhp, (void  **)&stmthp, OCI_HTYPE_STMT, 0, 0);

  OCIAttrSet((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)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 Topics**

- OCITransRollback()
  Rolls back the current transaction.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.7.2 OCITransDetach()

Detaches a global transaction.

### Purpose

Detaches a global transaction.

### Syntax

```
sword OCITransDetach ( OCISvcCtx    *svchp,
                       OCIError     *errhp,
                       ub4           flags );
```

### Parameters

**svchp (IN)**
The service context handle.

**errhp (IN)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 that 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 has no effect.

For example code demonstrating the use of `OCITransDetach()`, see the Examples section of `OCITransStart()`.

### Related Topics

- OCITransStart()
  Sets the beginning of a transaction.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.7.3 OCITransForget()

Causes the server to forget a heuristically completed global transaction.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 Topics

- OCITransCommit()
  Commits the transaction associated with a specified service context.
- OCITransRollback()
  Rolls back the current transaction.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.7.4 OCITransMultiPrepare()

Prepares a transaction with multiple branches in a single call.

### 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 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 Topics**

- OCITransPrepare()
  Prepares a global transaction for commit.

## 26.7.5 OCITransPrepare()

Prepares a global transaction for commit.

### Purpose

Prepares a global transaction for commit.

### Syntax

```
sword OCITransPrepare ( OCISvcCtx    *svchp,
                        OCIError     *errhp,
                        ub4           flags );
```

### Parameters

**svchp (IN)**
The service context handle.

**errhp (IN)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 indicates that the transaction is read-only. The `flags` parameter is not currently used.

### Related Topics

- OCITransCommit()
  Commits the transaction associated with a specified service context.
- OCITransForget()
  Causes the server to forget a heuristically completed global transaction.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.7.6 OCITransRollback()

Rolls back the current transaction.

### Purpose

Rolls back the current transaction.

### Syntax

```
sword OCITransRollback ( void          *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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 Examples section of OCITransCommit().

### Related Topics

- OCITransCommit()
  Commits the transaction associated with a specified service context.
- OCISessionBegin()
  Creates a user session and begins a user session for a given server.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.7.7 OCITransStart()

Sets the beginning of a transaction.

### 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 serializability 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` - Starts a read-only transaction.
- `OCI_TRANS_SERIALIZABLE` - Starts a serializable transaction.
- `OCI_TRANS_SEPARABLE` - Separates the transaction 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 code example and the one following that one demonstrate the use of OCI transactional calls for manipulating global transactions. The concept for a single session operating on different branches, as shown in the following code example, is illustrated by Figure 10-2.

Using OCITransStart() in a Single Session Operating on Different Branches

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  OCISession *usrhp;
  OCIStmt *stmthp1, *stmthp2;
  OCITrans *txnhp1, *txnhp2;
  void      *tmp;
  XID gxid;
  text sqlstmt[128];

  OCIEnvCreate(&envhp, OCI_DEFAULT, (void  *)0, 0, 0, 0,
        (size_t)0, (void  *)0);

  OCIHandleAlloc( (void  *) envhp, (void  **) &errhp, (ub4)
               OCI_HTYPE_ERROR, 52, (void  **) &tmp);
  OCIHandleAlloc( (void  *) envhp, (void  **) &srvhp, (ub4)
              OCI_HTYPE_SERVER, 52, (void  **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (void  *) envhp, (void  **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
               52, (void  **) &tmp);

  OCIHandleAlloc((void  *)envhp, (void  **)&stmthp1, OCI_HTYPE_STMT, 0, 0);
  OCIHandleAlloc((void  *)envhp, (void  **)&stmthp2, OCI_HTYPE_STMT, 0, 0);

  OCIAttrSet((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)srvhp, 0,
                 OCI_ATTR_SERVER, errhp);

  /* set the external name and internal name in server handle */
  OCIAttrSet((void  *)srvhp, OCI_HTYPE_SERVER, (void  *) "demo", 0,
                 OCI_ATTR_EXTERNAL_NAME, errhp);
  OCIAttrSet((void  *)srvhp, OCI_HTYPE_SERVER, (void  *) "txn demo", 0,
                         OCI_ATTR_INTERNAL_NAME, errhp);
```

```
/* allocate a user context handle */
OCIHandleAlloc((void  *)envhp, (void  **)&usrhp, (ub4) OCI_HTYPE_SESSION,
              (size_t) 0, (void  **) 0);

OCIAttrSet((void  *)usrhp, (ub4)OCI_HTYPE_SESSION, (void  *)"HR",
           (ub4)strlen("HR"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((void  *)usrhp, (ub4)OCI_HTYPE_SESSION, (void  *)"HR",
           (ub4)strlen("HR"),OCI_ATTR_PASSWORD, errhp);

OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, 0);

OCIAttrSet((void  *)svchp, (ub4)OCI_HTYPE_SVCCTX,
              (void  *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* allocate transaction handle 1 and set it in the service handle */
OCIHandleAlloc((void  *)envhp, (void  **)&txnhp1,  OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)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((void  *)txnhp1, OCI_HTYPE_TRANS, (void  *)&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((void  *)envhp, (void  **)&txnhp2,  OCI_HTYPE_TRANS, 0, 0);

OCIAttrSet((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)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((void  *)txnhp2, OCI_HTYPE_TRANS, (void  *)&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((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)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((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)txnhp2, 0,
                          OCI_ATTR_TRANS, errhp);
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

### Using OCITransStart() in a Single Session Operating on Multiple Branches Sharing the Same Transaction

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  OCISession *usrhp;
  OCIStmt *stmthp;
  OCITrans *txnhp1, *txnhp2;
  void      *tmp;
  XID gxid;
  text sqlstmt[128];

  OCIEnvCreate(&envhp, OCI_DEFAULT, (void  *)0, 0, 0, 0,
        (size_t)0, (void  *)0);

  OCIHandleAlloc( (void  *) envhp, (void  **) &errhp, (ub4) OCI_HTYPE_ERROR,
                52, (void  **) &tmp);
  OCIHandleAlloc( (void  *) envhp, (void  **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                52, (void  **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (void  *) envhp, (void  **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (void  **) &tmp);

  OCIHandleAlloc((void  *)envhp, (void  **)&stmthp, OCI_HTYPE_STMT, 0, 0);
```

```
OCIAttrSet((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)srvhp, 0,
                OCI_ATTR_SERVER, errhp);

/* set the external name and internal name in server handle */
OCIAttrSet((void  *)srvhp, OCI_HTYPE_SERVER, (void  *) "demo", 0,
                OCI_ATTR_EXTERNAL_NAME, errhp);
OCIAttrSet((void  *)srvhp, OCI_HTYPE_SERVER, (void  *) "txn demo2", 0,
                OCI_ATTR_INTERNAL_NAME, errhp);

/* allocate a user context handle */
OCIHandleAlloc((void  *)envhp, (void  **)&usrhp, (ub4) OCI_HTYPE_SESSION,
              (size_t) 0, (void  **) 0);

OCIAttrSet((void  *)usrhp, (ub4)OCI_HTYPE_SESSION, (void  *)"HR",
           (ub4)strlen("HR"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((void  *)usrhp, (ub4)OCI_HTYPE_SESSION, (void  *)"HR",
           (ub4)strlen("HR"),OCI_ATTR_PASSWORD, errhp);

OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, 0);

OCIAttrSet((void  *)svchp, (ub4)OCI_HTYPE_SVCCTX,
              (void  *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* allocate transaction handle 1 and set it in the service handle */
OCIHandleAlloc((void  *)envhp, (void  **)&txnhp1,  OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)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((void  *)txnhp1, OCI_HTYPE_TRANS, (void  *)&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((void  *)envhp, (void  **)&txnhp2,  OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)txnhp2, 0,
                        OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 123, 2] */
/* The global transaction is tightly coupled with earlier transactions */
/* 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((void  *)txnhp2, OCI_HTYPE_TRANS, (void  *)&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 returns */
  /* OCI_SUCCESS_WITH_INFO because all branches except the last branch */
  /* are treated as read-only transactions for tightly coupled transactions */

  OCIAttrSet((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)txnhp1, 0,
                        OCI_ATTR_TRANS, errhp);
  if (OCITransPrepare(svchp, errhp, (ub4) 0) == OCI_SUCCESS_WITH_INFO)
  {
    text errbuf[512];
    ub4 buflen;
    sb4 errcode;

    OCIErrorGet ((void  *) 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((void  *)svchp, OCI_HTYPE_SVCCTX, (void  *)txnhp2, 0,
                        OCI_ATTR_TRANS, errhp);
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

**Related Topics**

- OCITransDetach()
  Detaches a global transaction.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.8 Sharding Functions

Lists the sharding functions.

The following table lists the sharding OCI functions that are described in this section.

**Table 26-7 Sharding Functions**

| Function | Purpose |
|---|---|
| OCIShardingKeyColumnAdd() | Adds sharding key columns to form a compound sharding key or a compound super sharding key. |
| OCIShardingKeyReset() | Resets an already created sharding key or super sharding key for reuse with new key column values. |
| OCIShardInstancesGet() | Returns instance names for a given sharding key descriptor, super sharding key descriptor, and connection string. |

# 26.8.1 OCIShardingKeyColumnAdd()

Adds sharding key columns to form a compound sharding key or a compound super sharding key.

### Purpose

Adds sharding key columns to form a compound sharding key or a compound super sharding key. A sharding key column is a column in a table that is used as the key. If there are multiple columns to be used as the key then it is a compound key.

### Syntax

```
sword OCIShardingKeyColumnAdd(OCIShardingKey   *shardingKey,
                              OCIError         *errhp,
                              void             *col,
                              ub4               colLen,
                              ub2               colType,
                              ub4               mode);
```

### Parameters

**shardingKey (IN)**
The sharding key or super sharding key.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

**col (IN)**
The sharding key column, a component of the compound key. For character strings, the values are assumed to be in the client character set (specified by `NLS_LANG` or the `OCIEnvNLSCreate()` call).

**colLen (IN)**
Length of the sharding key column.

**colType (IN)**
The data type of the column.

**mode (IN)**
`OCI_DEFAULT` is the only value supported.

### Returns

`OCI_SUCCESS` or `OCI_ERROR`

### Usage Notes

Users must allocate an `OCIShardingKey` descriptor.

Call `OCIShardingKeyColumnAdd()` as many times as there are columns in the compound key (one time for a simple sharding key), in the order in which the key is defined in the database. The `colType` variable indicates the data type of the column.

The following table shows the supported OCI data type values for the `colType` parameter and its corresponding C data type.

| OCI Data Type | C Data Type |
|---|---|
| SQLT_NUM | ub1* |
| SQLT_CHR | OraText* |
| SQLT_DATE | ub1* |
| SQLT_TIMESTAMP | OCIDateTime* |
| SQLT_RAW | ub1* |
| SQLT_VNU | ub1* |
| SQLT_INT | int* |

### Examples

#### Example 26-1 Creating a Compound Sharding Key

Use the `OCIShardingKey` descriptor for the sharding key, then allocate a descriptor and call `OCIShardingKeyColumnAdd()` as many times as needed to create the compound sharding key.

```
OCIShardingKey   *shardKey;
text *name = “KK”;
int  empid = 150;

checker(&status, errhp, OCIDescriptorAlloc(envhp,(dvoid **)&shardKey,
           OCI_DTYPE_SHARDING_KEY, 0,(dvoid **)0)));

/* construct a compound key */

OCIShardingKeyColumnAdd(shardKey, OCIError *errhp,(ub4*)&empid,
sizeof(empid),
                    SQLT_INT, OCI_DEFAULT);
OCIShardingKeyColumnAdd(shardKey, OCIError *errhp, name, strlen(name),
                    SQLT_CHR, OCI_DEFAULT));
```

### Related Topics

- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.

- OCISessionPoolCreate()
  Initializes a session pool for use with OCI session pooling and database resident connection pooling (DRCP).

## 26.8.2 OCIShardingKeyReset()

Resets an already created sharding key or super sharding key for reuse with new key column values.

### Purpose

A sharding key or super sharding key descriptor can be reset using this routine, which removes all the key column values added prior to this call. After this call, the descriptor can be used to set new key column values.

### Syntax

```
sword OCIShardingKeyReset(OCIShardKey *shardKey,
                          OCIError    *errhp,
                          ub4          mode);
```

### Parameters

**shardKey(IN)**
The sharding key or super sharding key descriptor.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

**mode (IN)**
`OCI_DEFAULT` is the only value currently supported.

### Returns

`OCI_SUCCESS` or `OCI_ERROR`

### Usage Notes

None.

### Related Topics

- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIAttrGet()
  Gets the value of an attribute of a handle.

## 26.8.3 OCIShardInstancesGet()

For use with custom pools only, gets the instance names given a sharding key descriptor, super sharding key descriptor, and a connection string.

### Purpose

For use with custom pools only, returns instance names for a given sharding key descriptor for a chunk and super sharding key descriptor for a group of chunks and a connection string. Shards are locations in a set of databases where each database stores some part of the data. The part of the data stored at each database is represented by a set of *chunks*, where each chunk is associated with a certain range of the data.

### Syntax

```
sword OCIShardInstancesGet(
           void            **shTopoCtx,
           OCIError          *errhp,
           const OraText     *connstr,
           ub4                constrl,
           OCIShardingKey    *shardingKey,
           OCIShardingKey    *superShardingKey,
           OCIShardinst    ***shardinsts,
           ub4                numShardInsts,
           ub4                mode);
```

### Parameters

**shTopoCtx(IN/OUT)**
An opaque context that is created by OCI and returned in the first call to this method for a given connect string. This is an optional parameter and can be passed as `NULL`. For better performance, pass the `void * shardTopoCtx` returned from a prior call (if any) to `OCIShardInstancesGet()` for the same connection string.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

**connstr (IN)**
The original connect string passed by the application to the `OCISessionGet()` call.

**connstrl(IN)**
Length of the `connstr` parameter.

**shardingKey (IN)**
The sharding key for the desired chunk.

**superShardingKey (IN)**
The super sharding key for the desired chunk; this value can be NULL.

**shardInsts(OUT)**
An array of `OCIShardInst` descriptors corresponding to instances that serve the chunk associated with the sharding key and super sharding key.

**numShardInsts(OUT)**
The number of elements in `shardInsts`.

**mode(IN)**
`OCI_DEFAULT` is the only value supported.

### Returns

`OCI_SUCCESS` or `OCI_ERROR`

### Usage Notes

None.

### Examples

**Example 26-2 Custom Pool Example**

This example uses a custom pool and illustrates how to get the instance name of a given connection and how to look up for instances hosting a given sharding key, shard name, and super sharding name for a given connection. If there is no matching connection in the custom pool, it creates a new connection.

```
    OCIShardInstancesGet(
                       &shTopoCtx,
                       errhp,
                       connstr,
                       strlen(connstrl),
                            shardingKey,
                       superShardingKey,
                       &shardInsts,
                       &numShardInsts,
                       OCI_DEFAULT);

 /* Iterate through all the shard instances that have the data satisfying 
the sharding key. */

   for (k=0; k < numShardInsts; k++ )
   {
     OCIAttrGet(shardInstances[k],
               (ub4) OCI_DTYPE_SHARD_INST,
               (dvoid *)&iName,
               (ub4 *)&inameLen,
                OCI_ATTR_INSTNAME,
               (OCIError *)errhp);

/* Look up in the custom pool for a connection to the same shard instance. 
*/
/* The following attribute can be used to find the instance name. 
*/

   OCIAttrGet(svchp[i],
              OCI_HTYPE_SVCCTX,
              &cshardName[i],
              (ub4 *) &cshardNameLen[i],
              OCI_ATTR_INSTNAME, errhp);

/* If a matching connection is found (iName and cshardName[i]). */

    {
     OCIAttrSet(svchp[i],
               OCI_HTYPE_SVCCTX,
               shardingKey,
               sizeof(shardingKey),
               OCI_ATTR_SHARDING_KEY,
               errhp);

/* Return that connection to the requester. */
    return svchp[i];
   }
  }

/* If no free existing connection is found, create a new connection in the
pool and return it. */
  {
   OCIAttrSet(authp,
              OCI_HTYPE_AUTHINFO,
              shardingKey,
              sizeof(shardingKey),
              OCI_ATTR_SHARDING_KEY,
              errhp);

   OCISessionGet ((OCIEnv *)envhp,
                  (OCIError *)errhp,
                  (OCISvcCtx **)&mysvc,
                  (OCIAuthInfo *)authp,
                  (OraText *)connstr,
                  (ub4)strlen((char*)connstrl),
                  (OraText *)NULL, (ub4)0,
                  (OraText **)0, (ub4 *)0,
                  (boolean *)0,OCI_SESSGET_CUSTOM_POOL);
   return mysvc; /* Track in the pool too. */
  }
```

**Related Topics**

- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIAttrGet()
  Gets the value of an attribute of a handle.
- OCISessionGet()
  Gets a session.

# 26.9 Miscellaneous Functions

Lists and describes the miscellaneous OCI functions.

Table 26-8 lists the miscellaneous OCI functions that are described in this section.

**Table 26-8 Miscellaneous Functions**

| Function | Purpose |
|---|---|
| OCIBreak() | Perform an immediate asynchronous break |
| OCIClientVersion() | Return the client library version |
| OCIErrorGet() | Return error message and Oracle error |
| OCILdaToSvcCtx() | Toggle `Lda_Def` to service context handle |
| OCIPasswordChange() | Change password |
| OCIPing() | Confirm that the connection and the server are active |
| OCIReset() | Call after `OCIBreak()` to reset asynchronous operation and protocol |
| OCIRowidToChar() | Convert a Universal `ROWID` to character extended (base 64) representation |
| OCIServerRelease() | Get the Oracle release string |
| OCIServerVersion() | Get the Oracle version string |
| OCISvcCtxToLda() | Toggle service context handle to `Lda_Def` |
| OCIUserCallbackGet() | Identify the callback that is registered for handle |
| OCIUserCallbackRegister() | Register a user-created callback function |

## 26.9.1 OCIBreak()

Performs an immediate asynchronous break.

### Purpose

Performs an immediate (asynchronous) termination of any currently executing OCI function that is associated with a server.

### Syntax

```
sword OCIBreak ( void       *hndlp,
                 OCIError   *errhp );
```

### Parameters

**hndlp (IN/OUT)**
The service context handle or the server context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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 Linux or UNIX systems. `OCIBreak()` is the only OCI call allowed in a user signal handler.

> **Note:**
>
> `OCIBreak()` works 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.

**Related Topics**

- OCIReset()
  Resets the interrupted asynchronous operation and protocol.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Server Handle Attributes
  Lists and describes server handle attributes.
- Nonblocking Mode in OCI
  OCI provides the ability to establish a server connection in *blocking mode* or *nonblocking mode*.
- About Canceling Calls
  How do you cancel long-running or repeated OCI calls.

## 26.9.2 OCIClientVersion()

Returns the client library version.

**Purpose**

Returns the 5 digit Oracle Database version number of the client library at run time.

**Syntax**

```
void OCIClientVersion ( sword        *featureRelease,
                        sword        *releaseUpdate,
                        sword        *releaseUpdateRevision,
                        sword        *increment,
                        sword        *ext );
```

**Parameters**

**featureRelease (OUT)**
The feature release year.

**releaseUpdate (OUT)**
The release update number.

**releaseUpdateRevision (OUT)**
The release update revision number.

**increment (OUT)**
The release update increment number.

**ext (OUT)**
The extension number.

### Comments

Beginning with release 18c, version 18.1, there are five new client macros for extracting each of the components of the database version. These macros are intended to be used in conjunction with the encoded database version number returned by `OCIServerRelease2()` to extract each of the 5 components of the database version. These macros are named according to the new release naming scheme started with version 18.1. The 5 digits extracted from the version number are: `<feature release>.<release update>.<release update revision>.<release update increment>.<extension>`. For example, 18.1.1.1.1 would represent feature release 18, release update 1, release update revision 1, release update increment 1, extension 1.`featureRelease()` returns the feture release year of OCI client that the application is running with. This is useful for the application to know at run time. 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 level of the client patchset.

These macros are useful for writing a generic application that can be built and run with different versions of OCI client. For example:

```
....
#if (featureRelease > 12)
...
#endif
....
```

Even though the naming is based on the new convention, for convenience, the same macros can be used to extract the corresponding 5 components of the database version from an encoded version number returned by `OCIServerRelease()` from a pre version 18.1 database, which used a different encoding scheme. In that case though, the components correspond to the following:

- Version number
- Release number
- Update number
- Porting release number
- Porting update number

### Related Topics

- OCIServerRelease()
  Returns the Oracle Database release string.

## 26.9.3 OCIErrorGet()

Returns an error message and an Oracle Database error code.

**Purpose**

Returns an error message in the buffer provided and an Oracle Database error code.

**Syntax**

```
sword OCIErrorGet ( void       *hndlp,
                    ub4         recordno,
                    OraText    *sqlstate,
                    sb4        *errcodep,
                    OraText    *bufp,
                    ub4         bufsiz,
                    ub4         type );
```

**Parameters**

**hndlp (IN)**
The error handle, usually, or the environment handle (for errors on `OCIEnvCreate()`, `OCIHandleAlloc()`).

**recordno (IN)**
Indicates the status record from which the application seeks information. 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`.
Use one of the following constants to define the error message buffers into which you get the returned message back from `OCIErrorGet()`:

```
# define OCI_ERROR_MAXMSG_SIZE 1024 /* max size of an error message */
# define OCI_ERROR_MAXMSG_SIZE2 3072 /* new length max size of an error message */
```

You should use `OCI_ERROR_MAXMSG_SIZE2` to ensure you get more information in the returned error text.
For example, you can do the following:

```
char errorMesg[OCI_ERROR_MAXMSG_SIZE2];
```

Then pass this buffer into `OCIErrorGet()`. You also need to pass the same `OCI_ERROR_MAXMSG_SIZE2` value into the `OCIErrorGet()` call to indicate the size of the buffer that you have allocated.

**type (IN)**
The type of the handle (`OCI_HTYPE_ERROR` or `OCI_HTYPE_ENV`).

### Comments

This function does not support SQL statements. Usually, `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.

Note that if `OCIErrorGet()` is called following an OCI call that does not return an `OCI_ERROR` or `OCI_SUCCESS_WITH_INFO`, then an error message from some earlier OCI call that resulted in an error may be found by `OCIErrorGet()`. To avoid this issue, `OCIErrorGet()` should only be called following OCI calls that return either `OCI_ERROR` or `OCI_SUCCESS_WITH_INFO`.

The error handle is originally allocated with a call to `OCIHandleAlloc()`.

**Note:**

The `OCIErrorGet()` function returns at least one single diagnostic record. Multiple diagnostic records can be retrieved by calling the `OCIErrorGet()` method repeatedly, until there are no more records and the method returns `OCI_NO_DATA`.

**See Also:**

Error Handling in OCI

### Example

The following code example shows a simplified example of a function for error checking using `OCIErrorGet()`.

Using OCIErrorGet() for Error Checking

```
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 ((void  *) 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 ((void  *) 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 Topics**

- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 26.9.4 OCILdaToSvcCtx()

Converts a V7 `Lda_Def` to a V8 or later service context handle.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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.

You should use the `OCILdaToSvcCtx()` call only for resetting an `Lda_Def` obtained from `OCISvcCtxToLda()` back to a service context handle. It cannot be used to transform an `Lda_def` that 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 can 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.

### Related Topics

- OCISvcCtxToLda()
  Toggles between a V8 or later service context handle and a V7 `Lda_Def`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Handler and Descriptor Attributes
  Describes the attributes for OCI handles and descriptors.

## 26.9.5 OCIPasswordChange()

Changes the password of an account.

### Purpose

Allows the password of an account to be changed.

### Syntax

```
sword OCIPasswordChange ( OCISvcCtx     *svchp,
                          OCIError      *errhp,
                          const OraText *user_name,
                          ub4            usernm_len,
                          const OraText *opasswd,
                          ub4            opasswd_len,
                          const OraText *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 that you can pass to `OCIErrorGet()` for diagnostic information when there is 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. The `usernm_len` value 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. The `opasswd_len` value 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, a call with this flag 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 created, a call with this flag only changes the password and has no effect on the session. Hence the user still remains logged in.

  The `OCI_AUTH` mode can be used with any of the `OCI_CPW_*` modes listed in this section to establish the respective administrative session for the expired user account at logon before changing the password.

- `OCI_CPW_SYSDBA` — In this mode, you are authenticated for `SYSDBA` access.
- `OCI_CPW_SYSOPER` — In this mode, you are authenticated for `SYSOPER` access.
- `OCI_CPW_SYSASM` — In this mode, you are authenticated for `SYSASM` access.
- `OCI_CPW_SYSBKP` — In this mode, you are authenticated for `SYSBKP` access.
- `OCI_CPW_SYSDGD` — In this mode, you are authenticated for `SYSDGD` access.
- `OCI_CPW_SYSKMT` — In this mode, you are authenticated for `SYSKMT` access.

### 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 established, this call authenticates the account using the old password and then changes the password to the new password.
- If the user session is not established, this call establishes a user session and authenticates the account using the old password, and 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.

For a Release 12.1 or later client password change with a Release 11.2 server, you must first call `OCISessionBegin()` before calling `OCIPasswordChange()`; otherwise the password change operation fails with an `ORA-1017` error.

### Related Topics

- OCISessionBegin()
  Creates a user session and begins a user session for a given server.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.9.6 OCIPing()

Confirms that the connection and the server are active.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**mode (IN)**
The mode for the call. Use `OCI_DEFAULT`.

### Comments

`OCIPing()` 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.

You can use `OCIPing()` to make a lightweight call to the server. A successful return of the call indicates that 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. Because `OCIPing()` is a round-trip call, you can also use it 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. The call is useful when the application requires the back-end cursors to be closed immediately, which otherwise would be closed in the next round-trip on that connection. Also, with the enhanced functionality of `OCI_ATTR_SERVER_STATUS`, OCIPing may not be needed if the requirement is just to check the health of the connection. See the `OCI_ATTR_SERVER_STATUS` attribute in Server Handle Attributes for more information.

### Related Topics

- OCIHandleFree()
  Explicitly deallocates a handle
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.9.7 OCIReset()

Resets the interrupted asynchronous operation and protocol.

### Purpose

Must be called if an `OCIBreak()` call was issued while a nonblocking operation was in progress.

### Syntax

```
sword OCIReset ( void        *hndlp,
                 OCIError    *errhp );
```

### Parameters

**hndlp (IN)**
The service context handle or the server context handle.

**errhp (IN)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

### Comments

This call is called in nonblocking mode only. It resets the interrupted asynchronous operation and protocol. OCIReset() must be called if an `OCIBreak()` call was issued while a nonblocking operation was in progress.

### Related Topics

- OCIBreak()
  Performs an immediate asynchronous break.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.9.8 OCIRowidToChar()

Converts a Universal `ROWID` to character extended (base 64) representation.

### 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 that 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.
If there is truncation during conversion, `outbfp` contains the length required to make conversion successful. An error is also returned.

**errhp (IN)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is 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`.

If the environment was created using `OCIEnvNlsCreate()` with parameters `charset` and `ncharset` set to `OCI_UTF16ID`, the function `OCIRowidToChar()` returns the rowid representation in ASCII, not in UTF-16 as expected.

### Related Topics

- OCIDescriptorAlloc()
  Allocates storage to hold descriptors or LOB locators.
- OCIBindByPos()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIBindByPos2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this call when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIBindByName()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.
- OCIBindByName2()
  Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block. Use this function when working with data types when actual lengths exceed `UB2MAXVAL` on the client.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.9.9 OCIServerRelease()

Returns the Oracle Database release string.

### Purpose

Returns the Oracle Database release string.

### Syntax

```
sword OCIServerRelease ( void         *hndlp,
                         OCIError     *errhp,
                         OraText      *bufp,
                         ub4           bufsz
                         ub1           hndltype
                         ub4          *version );
```

### Parameters

**hndlp (IN)**
The service context handle or the server context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**bufp (IN/OUT)**
The buffer in which the release string is returned.

**bufsz (IN)**
The length of the buffer in number of bytes.

**hndltype (IN)**
The type of handle passed to the function.

**version (IN/OUT)**
The release string in integer format.

### Comments

The buffer pointer `bufp` points to the release information in a string representation up to the `bufsz` including the `NULL` terminator. If the buffer size is too small, the result is truncated to the size `bufsz`. The `version` argument contains the 5-digit Oracle Database release string in integer format, which can be retrieved using the following macros:

```
#define MAJOR_NUMVSN(v) ((sword)(((v) >> 24) & 0x000000FF))      /* version number */
#define MINOR_NUMRLS(v) ((sword)(((v) >> 20) & 0x0000000F))      /* release number */
#define UPDATE_NUMUPD(v) ((sword)(((v) >> 12) & 0x000000FF))     /* update number */
#define PORT_REL_NUMPRL(v) ((sword)(((v) >> 8) & 0x0000000F))    /* port release number */
#define PORT_UPDATE_NUMUPP(v) ((sword)(((v) >> 0) & 0x000000FF)) /* port update number */
```

### Related Topics

- OCIServerVersion()
  Returns the Oracle Database version string.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.9.10 OCIServerRelease2()

Returns the Oracle Database release string.

### Purpose

Returns the Oracle Database release string.

### Syntax

```
sword OCIServerRelease2( void          *hndlp,
                         OCIError      *errhp,
                         OraText       *bufp,
                         ub4            bufsz,
                          ub1           hndltype,
                          ub4          *versionp,
                          ub4           mode );
```

### Parameters

**`hndlp (IN)`**
The service context handle or the server context handle.

**`errhp (IN/OUT)`**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**`bufp (IN/OUT)`**
The buffer in which the release string is returned.

**`bufsz (IN)`**
The length of the buffer in number of bytes.

**`hndltype (IN)`**
The type of handle passed to the function.

**`versionp (IN/OUT)`**
The release string in integer format.

**`mode (IN)`**
The only valid value is `OCI_DEFAULT`.

### Comments

The buffer pointer `bufp` points to the release information in a string representation up to the `bufsz` including the `NULL` terminator. If the buffer size is too small, the result is truncated to the size `bufsz`. The `version` argument contains the 5-digit Oracle Database release string in integer format, which can be retrieved using the following macros:

```
OCI_SERVER_RELEASE_REL(v)
             /* old: version number */
             /* new: feature release */
OCI_SERVER_RELEASE_REL_UPD(v)
            /* old: release number */
            /* new: release update */
OCI_SERVER_RELEASE_REL_UPD_REV(v)
           /* old: update number */
           /* new: release update revision */
OCI_SERVER_RELEASE_REL_UPD_INC(v)
          /* old: porting release number */
          /* new: release update increment */
OCI_SERVER_RELEASE_EXT(v)
         /* old: porting update number */
         /* new: extension */
```

Old means applicable when connected to a pre-version 18.1 database. New means applicable when connected to version 18.1 or later database.

## 26.9.11 OCIServerVersion()

Returns the Oracle Database version string.

### Purpose

Returns the Oracle Database version string.

### Syntax

```
sword OCIServerVersion ( void         *hndlp,
                         OCIError     *errhp,
                         OraText      *bufp,
                         ub4           bufsz
                         ub1           hndltype );
```

### Parameters

**hndlp (IN)**
The service context handle or the server context handle.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**bufp (IN/OUT)**
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 Oracle Database. 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 Topics

- OCIClientVersion()
  Returns the client library version.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 26.9.12 OCISvcCtxToLda()

Toggles between a V8 or later service context handle and a V7 `Lda_Def`.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**ldap (IN/OUT)**
A Logon Data Area for Oracle7-style OCI calls that 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 that 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.

### Related Topics

- OCILdaToSvcCtx()
  Converts a V7 `Lda_Def` to a V8 or later service context handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Handler and Descriptor Attributes
  Describes the attributes for OCI handles and descriptors.

## 26.9.13 OCIUserCallbackGet()

Determines the callback that is registered for a handle.

### Purpose

Determines the callback that is registered for a handle.

### Syntax

```
sword OCIUserCallbackGet ( void    *hndlp,
                           ub4      type,
                           void    *ehndlp,
                           ub4      fcode,
                           ub4      when,
                           OCIUserCallback (*callbackp)
                                           (
                                             void   *ctxp,
                                             void   *hndlp,
                                             ub4     type,
                                             ub4     fcode,
                                             ub1     when,
                                             sword   returnCode,
                                             ub4    *errnop,
                                             va_list arglist
                                            ),
                           void    **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 26-9.

**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 are not called.

Instead, processing jumps to the exit callbacks. For information about this parameter, see `OCIUserCallbackRegister()`.

**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:**
>
> `OCIUserCallbackRegister()` for information about the parameters of `callbackp`

**ctxpp (OUT)**
A pointer to return context for the currently registered callback.

**ucbDesc (IN)**
A descriptor provided by OCI. This descriptor is passed by OCI in the environment callback. It contains the priority at which the callback would be registered. 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 discovers or detects what callback is registered for a particular handle.

### Related Topics

- OCIUserCallbackRegister()
  Registers a user-created callback function.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Restrictions on Callback Functions
  Details the restrictions on callback functions.

## 26.9.14 OCIUserCallbackRegister()

Registers a user-created callback function.

### Purpose

Registers a user-created callback function.

### Syntax

```
sword OCIUserCallbackRegister ( void    *hndlp,
                                ub4      type,
                                void    *ehndlp,
                                OCIUserCallback  (callback)
                                                 (
                                                   void    *ctxp,
                                                   void    *hndlp,
                                                   ub4      type,
                                                   ub4      fcode,
                                          ub1       when,
                                          sword     returnCode,
                                          ub4      *errnop,
                                          va_list   arglist
                                        ),
                      void      *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()`. Because an error handle is not available within `OCIEnvCallback`, 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 deregister 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 26-9.

**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 the callback 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)**
A descriptor provided by OCI. This descriptor is passed by OCI in the environment callback. It contains the priority at which the callback would be registered. 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 with the OCI environment.

Such callbacks allow an application to:

- Trace OCI calls for debugging and performance measurements
- Perform additional pre-processing or post-processing after selected OCI calls
- Substitute the body of a given function with proprietary code to execute on a foreign data source

The OCI supports: *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:

1. Execute entry callbacks.
2. Execute replacement callbacks.
3. Execute OCI code.
4. 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 determine the callback that is registered for the handle, you can use `OCIUserCallbackGet()`.

The prototype of the `OCIUserCallback` typedef is:

```
typedef sword (*OCIUserCallback)
           (void    *ctxp,
            void    *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 only a type of `OCI_HTYPE_ENV` is allowed, 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 26-9. Note that callbacks can be registered for only the OCI calls listed in Table 26-3.

**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` is always 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 `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, however, the previous callback returned `OCI_CONTINUE`, then whatever value 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 dereferenced using `va_arg`, as illustrated in the user callback demonstration programs.

**See Also:**

OCI Demonstration Programs

Table 26-9 and Table 26-10 list the OCI Function codes and provides the OCI routine name and its function number.

### Table 26-9 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 | 74 | OCILobFileSetName |
| 11 | OCISessionEnd | 43 | OCILobAppend | 75 | OCILobFileGetName |
| 12 | OCIPasswordChange | 44 | OCILobErase | 76 | OCILogon |
| 13 | OCIStmtPrepare | 45 | OCILobGetLength | 77 | OCILogoff |
| 14 | (not used) | 46 | OCILobTrim | 78 | OCILobDisableBuffering |
| 15 | (not used) | 47 | OCILobRead | 79 | OCILobFlushBuffer |
| 16 | (not used) | 48 | OCILobWrite | 80 | OCILobLoadFromFile |
| 17 | OCIBindDynamic | 49 | (not used) | 81 | OCILobOpen |
| 18 | OCIBindObject | 50 | OCIBreak | 82 | OCILobClose |
| 19 | (not used) | 51 | OCIServerVersion | 83 | OCILobIsOpen |
| 20 | OCIBindArrayOfStruct | 52 | (not used) | 84 | OCILobFileIsOpen |
| 21 | OCIStmtExecute | 53 | (not used) | 85 | OCILobFileExists |
| 22 | (not used) | 54 | OCIAttrGet | 86 | OCILobFileCloseAll |
| 23 | (not used) | 55 | OCIAttrSet | 87 | OCILobCreateTemporary |
| 24 | (not used) | 56 | OCIParamSet | 88 | OCILobFreeTemporary |
| 25 | OCIDefineObject | 57 | OCIParamGet | 89 | OCILobIsTemporary |
| 26 | OCIDefineDynamic | 58 | OCIStmtGetPieceInfo | 90 | OCIAQEnq |
| 27 | OCIDefineArrayOfStruct | 59 | OCILdaToSvcCtx | 91 | OCIAQDeq |
| 28 | OCIStmtFetch | 60 | (not used) | 92 | OCIReset |
| 29 | OCIStmtGetBindInfo | 61 | OCIStmtSetPieceInfo | 93 | OCISvcCtxToLda |
| 30 | (not used) | 62 | OCITransForget | 94 | OCILobLocatorAssign |
| 31 | (not used) | 63 | OCITransPrepare | 95 | (not used) |

**Table 26-9 (Cont.) OCI Function Codes**

| # | OCI Routine | # | OCI Routine | # | OCI Routine |
|---|---|---|---|---|---|
| 32 | OCIDescribeAny | 64 | OCITransRollback | 96 | OCIAQListen |

**Table 26-10 Continuation of OCI Function Codes from 97 and Higher**

| # | OCI Routine | # | OCI Routine | # | OCI Routine |
|---|---|---|---|---|---|
| 97 | Reserved | 113 | OCILobErase2 | 129 | OCILobGetOptions |
| 98 | Reserved | 114 | OCILobGetLength2 | 130 | OCILobSetOptions |
| 99 | OCITransMultiPrepare | 115 | OCILobLoadFromFile2 | 131 | OCILobFragementInsert |
| 100 | OCIConnectionPoolCreate | 116 | OCILobRead2 | 132 | OCILobFragementDelete |
| 101 | OCIConnectionPoolDestroy | 117 | OCILobTrim2 | 133 | OCILobFragementMove |
| 102 | OCILogon2 | 118 | OCILobWrite2 | 134 | OCILobFragementReplace |
| 103 | OCIRowidToChar | 119 | OCILobGetStorageLimit | 135 | OCILobGetDeduplicateRegions |
| 104 | OCISessionPoolCreate | 120 | OCIDBStartup | 136 | OCIAppCtxSet |
| 105 | OCISessionPoolDestroy | 121 | OCIDBShutdown | 137 | OCIAppCtxClearAll |
| 106 | OCISessionGet | 122 | OCILobArrayRead | 138 | OCILobGetContentType |
| 107 | OCISessionRelease | 123 | OCILobArrayWrite | 139 | OCILobSetContentType |
| 108 | OCIStmtPrepare2 | 124 | OCIAQEnqStreaming | | |
| 109 | OCIStmtRelease | 125 | OCIAQGetReplayInfo | | |
| 110 | OCIAQEnqArray | 126 | OCIAQResetReplayInfo | | |
| 111 | OCIAQDeqArray | 127 | OCIArrayDescriptorAlloc | | |
| 112 | OCILobCopy2 | 128 | OCIArrayDescriptorFree | | |

**Related Topics**

- OCIUserCallbackGet()
  Determines the callback that is registered for a handle.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 27

# OCI Navigational and Type Functions

This chapter describes the OCI navigational functions that are used to navigate through objects retrieved from an Oracle database.

It also contains the descriptions of the functions that are used to obtain type descriptor objects (TDOs).

> **See Also:**
>
> For code examples, see the demonstration programs included with your Oracle Database installation. For additional information, see OCI Demonstration Programs.

This chapter contains these topics:

- Introduction to the Navigational and Type Functions
- OCI Flush or Refresh Functions
- OCI Mark or Unmark Object and Cache Functions
- OCI Get Object Status Functions
- OCI Miscellaneous Object Functions
- OCI Pin, Unpin, and Free Functions
- OCI Type Information Accessor Functions

## 27.1 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. OCI provides a navigational interface to objects in an Oracle database. Those calls are described in this chapter.

The OCI object environment is initialized when the application calls `OCIEnvCreate()`, `OCIEnvNlsCreate()`, or `OCIInitialize()` (deprecated) in `OCI_OBJECT` mode.

> **See Also:**
>
> OCI Object-Relational Programming and Object Advanced Topics in OCI for more information about using the calls in this chapter

## 27.1.1 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. 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 that 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 that 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.

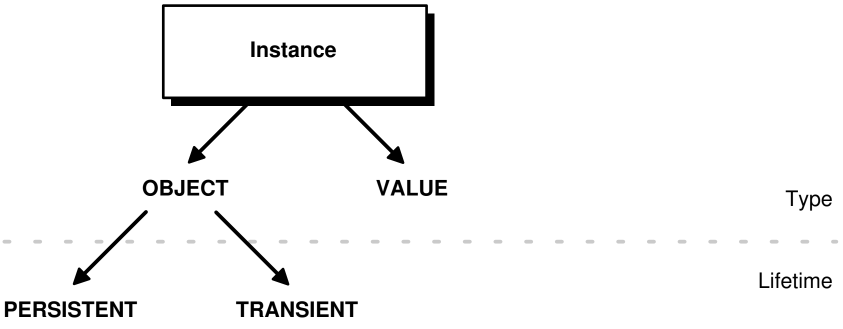
A value can be standalone or embedded. A standalone value is usually obtained by issuing a select statement. OCI also allows the client program to select a row of object table into a value by issuing a SQL statement. A referenceable object in the database can be represented as a value that cannot be 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 inline attribute in an object (such as number or nested object), or an inline element in a collection.

All values are considered to be transient by OCI, which means that OCI does not support automatic flushing of a value to the database, and the client must explicitly execute a SQL statement to store a value into the database. Embedded values are flushed when their containing instances are flushed.

Figure 27-1 shows how instances can be classified according to their type and lifetime. The type can be an object or the value of an object. The lifetime can be persistent (can exist longer than the duration of a session) or transient (can exist no long than the duration of the session).

**Figure 27-1 Classification of Instances by Type and Lifetime**



The distinction between various instances is further described in Table 27-1.

**Table 27-1 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 |

## 27.1.2 Terminology

Describes the terminology used.

The remainder of this chapter uses the following terms:

- An *object* is 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*), marked as updated, or marked as deleted.

> **See Also:**
>
> Persistent Objects, Transient Objects, and Values for further discussion of the terms used to refer to different types of objects

## 27.1.3 Conventions for OCI Functions

Describes the conventions for OCI functions.

See the Conventions for OCI Functions for the conventions used in describing each function. The entries for each function may also contain the following information:

## 27.1.4 Return Values

Describes the return values by the function if the function returns something other then the standard codes.

A description of what value is returned by the function if the function returns something other then the standard codes listed in Table 27-3.

## 27.1.5 Navigational Function Return Values

Describes the navigational function return values.

Table 27-2 lists the values that OCI navigational functions typically return.

**Table 27-2 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:**
>
> Error Handling in OCI for more information about return codes and error handling

## 27.1.6 Server Round-Trips for Cache and Object Functions

Describes the number of server round-trips required for individual OCI cache and object functions.

For a table showing the number of server round-trips required for individual OCI cache and object functions, see Table C-3.

## 27.1.7 Navigational Function Error Codes

Describes the external Oracle error codes that can be returned by each of the OCI navigational functions.

Table 27-3 lists the external Oracle error codes that can be returned by each of the OCI navigational functions. The list following the table identifies what each error represents.

**Table 27-3 OCI Navigational Functions Error Codes**

| Function | Possible ORA Errors |
|---|---|
| `OCICacheFlush()` | 24350, 21560, 21705 |
| `OCICacheFree()` | 24350, 21560, 21705 |
| `OCICacheRefresh()` | 24350, 21560, 21705 |

### Table 27-3 (Cont.) OCI Navigational Functions Error Codes

| Function | Possible ORA Errors |
|---|---|
| `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 |
| `OCIObjectGetObjectRef()` | 24350, 21560, 21710 |
| `OCIObjectGetTypeRef()` | 24350, 21560, 21710 |
| `OCIObjectIsDirty()` | 24350, 21560, 21710 |
| `OCIObjectIsLocked()` | 24350, 21560, 21710 |
| `OCIObjectLock()` | 24350, 21560, 21701, 21708, 21710 |
| `OCIObjectLockNoWait()` | 24350, 21560, 21701, 21708, 21710 |
| `OCIObjectMarkDelete()` | 24350, 21560, 21700, 21701, 21702, 21710 |
| `OCIObjectMarkDeleteByRef()` | 24350, 21560 |
| `OCIObjectMarkUpdate()` | 24350, 21560, 21700, 21701, 21710 |
| `OCIObjectNew()` | 24350, 21560, 21705, 21710 |
| `OCIObjectPin()` | 24350, 21560, 21700, 21702 |
| `OCIObjectPinCountReset()` | 24350, 21560, 21710 |
| `OCIObjectPinTable()` | 24350, 21560, 21705 |
| `OCIObjectRefresh()` | 24350, 21560, 21709, 21710 |
| `OCIObjectSetAttr()` | 21560, 21600, 22305, 22279, 21601 |
| `OCIObjectUnmark()` | 24350, 21560, 21710 |
| `OCIObjectUnmarkByRef()` | 24350, 21560 |
| `OCIObjectUnpin()` | 24350, 21560, 21710 |

The ORA errors in Table 27-3 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 - cannot 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

# 27.2 OCI Flush or Refresh Functions

Lists and describes OCI flush or refresh functions.

Table 27-4 describes the OCI flush or refresh functions that are described in this section.

**Table 27-4 Flush or Refresh Functions**

| Function | Purpose |
|---|---|
| OCICacheFlush() | Flush modified persistent objects in cache to server |
| OCICacheRefresh() | Refresh pinned persistent objects |
| OCIObjectFlush() | Flush a modified persistent object to the server |
| OCIObjectRefresh() | Refresh a persistent object |

## 27.2.1 OCICacheFlush()

Flushes modified persistent objects to the server.

### Purpose

Flushes modified persistent objects to the server.

### Syntax

```
sword OCICacheFlush ( OCIEnv               *env,
                      OCIError             *err,
                      const OCISvcCtx      *svc,
                      void                 *context,
                      OCIRef               *(*get)
                                           ( void     *context,
                                             ub1      *last ),
                      OCIRef              **ref );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 that acts as an iterator to retrieve a batch of dirty objects that need to be flushed. If the function is not `NULL`, this function is 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 signaled. 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`) points to the object that is causing the error. If `ref` is `NULL`, then the object is not returned. If `*ref` is `NULL`, then a reference is 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 objects, 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 new or marked as updated or marked as deleted.

**See Also:**

`OCIObjectFlush()`

This function incurs, at most, one network round-trip.

**Related Topics**

- OCIObjectFlush()
  Flushes a modified persistent object to the server.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.2.2 OCICacheRefresh()

Refreshes all pinned persistent objects in the cache.

**Purpose**

Refreshes all pinned persistent objects in the cache.

**Syntax**

```
sword OCICacheRefresh ( OCIEnv            *env,
                       OCIError          *err,
                       const OCISvcCtx   *svc,
                       OCIRefreshOpt      option,
                       void              *context,
                       OCIRef            *(*get)(void  *context),
                       OCIRef           **ref );
```

**Parameters**

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the description of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 ignores 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 that acts as an iterator to retrieve a batch of objects that need to be refreshed. If the function is not `NULL`, this function is called to get a reference of an object. If the reference is not `NULL`, then the object is 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`) points to the object that is causing the error. If `ref` is `NULL`, then the object is not returned. If `*ref` is `NULL`, then a reference is 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 objects, the given `ref` is initialized to be a `NULL` reference (`OCIRefIsNull(*ref)` is `TRUE`).

### Comments

This function refreshes all pinned persistent objects and frees all unpinned persistent objects from the object cache.

> **Note:**
>
> 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 that were saved prior to this call may be invalidated. Examples of attributes using secondary-level memory include `OCIString *`, `OCIColl *`, and `OCIRaw *`.

### Related Topics

- OCIObjectRefresh()
  Refreshes a persistent object from the most current database snapshot.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- About Refreshing an Object Copy
  When refreshed, an object copy is reloaded with the latest value of the corresponding object in the server.

## 27.2.3 OCIObjectFlush()

Flushes a modified persistent object to the server.

### Purpose

Flushes a modified persistent object to the server.

### Syntax

```
sword OCIObjectFlush ( OCIEnv        *env,
                       OCIError      *err,
                       void          *object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 this function writes the object 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()` operation 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.

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.
- OCICacheFlush()
  Flushes modified persistent objects to the server.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectCopy()
  Copies a source instance to a destination.
- OCILobAssign()
  Assigns one LOB or `BFILE` locator to another
- OCILobLocatorAssign()
  Assigns one LOB or `BFILE` locator to another.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- LOB Functions

## 27.2.4 OCIObjectRefresh()

Refreshes a persistent object from the most current database snapshot.

### Purpose

Refreshes a persistent object from the most current database snapshot.

### Syntax

```
sword OCIObjectRefresh ( OCIEnv        *env,
                         OCIError      *err,
                         void          *object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 in 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.

**Note:**

Modifications made to objects (dirty objects) since the last flush are lost if unmarked objects are refreshed by this function.

Table 27-5 shows how the various meta-attribute flags and durations of an object are modified after being refreshed.

**Table 27-5 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 is *replaced-in-place*. When an object is replaced-in-place, the top-level memory of the object is 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 are freed and reallocated.

You should be careful when writing functionality that retains a pointer to the secondary memory chunk, such as assigning the address of a secondary memory chunk to a local variable, because this pointer can become invalid after the object is refreshed.

This function does no affect transient objects or values.

**Related Topics**

- OCICacheRefresh()
  Refreshes all pinned persistent objects in the cache.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 27.3 OCI Mark or Unmark Object and Cache Functions

Lists and describes the OCI mark or unmark object and cache functions.

Table 27-6 describes the OCI mark or unmark object and cache functions that are described in this section.

**Table 27-6    Mark or Unmark Object and Cache Functions**

| Function | Purpose |
|---|---|
| OCICacheUnmark() | Unmark objects in the cache |
| OCIObjectMarkDelete() | Mark an object deleted or delete a value instance |
| OCIObjectMarkDeleteByRef() | Mark an object deleted when given a reference to it |
| OCIObjectMarkUpdate() | Mark an object as updated or dirty |
| OCIObjectUnmark() | Unmark an object |
| OCIObjectUnmarkByRef() | Unmark an object, when given a reference to it |

## 27.3.1 OCICacheUnmark()

Unmarks all dirty objects in the object cache.

### 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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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.

**Related Topics**

- OCIObjectUnmark()
  Unmarks an object as dirty.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> `OCIObjectUnmark()` for more information about unmarking an object

## 27.3.2 OCIObjectMarkDelete()

Marks a standalone instance as deleted, when given a pointer to the instance.

**Purpose**

Marks a standalone instance as deleted, when given a pointer to the instance.

**Syntax**

```
sword OCIObjectMarkDelete ( OCIEnv        *env,
                            OCIError      *err,
                            void          *instance );
```

**Parameters**

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 Topics

- OCIObjectMarkDeleteByRef()
  Marks an object as deleted, when given a reference to the object.
- OCIObjectGetProperty()
  Retrieves a given property of an object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.3.3 OCIObjectMarkDeleteByRef()

Marks an object as deleted, when given a reference to the object.

### Purpose

Marks an object as deleted, when 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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 Topics

- OCIObjectMarkDelete()
  Marks a standalone instance as deleted, when given a pointer to the instance.
- OCIObjectGetProperty()
  Retrieves a given property of an object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.3.4 OCIObjectMarkUpdate()

Marks a persistent object as updated (dirty).

### Purpose

Marks a persistent object as updated (*dirty*).

### Syntax

```
sword OCIObjectMarkUpdate ( OCIEnv        *env,
                            OCIError      *err,
                            void          *object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 (*dirty*). The following special rules apply to different types of objects. The *dirty* status of an object can be checked by calling `OCIObjectIsLocked()`.

**For Persistent Objects**
This function marks the specified persistent object as updated.
When the object cache is flushed, it writes the persistent objects to the server. 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 being flushed.

**For Transient Objects**
This function marks the specified transient object as updated. The transient objects are not written to the server. It is an error to update a deleted object.

**For Values**
This function has no effect on values.

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.
- OCIObjectGetProperty()
  Retrieves a given property of an object.
- OCIObjectIsDirty()
  Checks to see if an object is marked as dirty.
- OCIObjectUnmark()
  Unmarks an object as dirty.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

- OCIObjectIsLocked()
  Gets lock status of an object.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- About Marking Objects and Flushing Changes
  The application must take specific steps to ensure that changes to objects by marking them and then flushing them are written in the database.

## 27.3.5 OCIObjectUnmark()

Unmarks an object as dirty.

### Purpose

Unmarks an object as dirty.

### Syntax

```
sword OCIObjectUnmark ( OCIEnv       *env,
                        OCIError     *err,
                        void         *object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 are not written to the server. If the object is marked as locked, it remains marked as locked. The changes that have already been made to the object are not undone implicitly.

**For Values**
This function has no effect if called on a value.

### Related Topics

- OCIObjectUnmarkByRef()
  Unmarks an object as dirty, when given a `ref` to the object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.3.6 OCIObjectUnmarkByRef()

Unmarks an object as dirty, when given a `ref` to the object.

### Purpose

Unmarks an object as dirty, when 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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 are not written to the server. If the object is marked as locked, it remains marked as locked. The changes that have already been made to the object are not undone implicitly.

**For Values**
This function has no effect on values.

### Related Topics

- OCIObjectUnmark()
  Unmarks an object as dirty.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.4 OCI Get Object Status Functions

Lists and describes the OCI get object status functions.

Table 27-7 describes the OCI get object status functions that are described in this section.

**Table 27-7 Get Object Status Functions**

| Function | Purpose |
|---|---|
| OCIObjectExists() | Get the existent status of an instance |
| OCIObjectGetProperty() | Get the status of a particular object property |
| OCIObjectIsDirty() | Get the dirtied status of an instance |
| OCIObjectIsLocked() | Get the locked status of an instance |

### 27.4.1 OCIObjectExists()

Returns the existence meta-attribute of a standalone instance.

**Purpose**

Returns the existence meta-attribute of a standalone instance.

**Syntax**

```
sword OCIObjectExists ( OCIEnv         *env,
                        OCIError       *err,
                        void           *ins,
                        boolean        *exist );
```

**Parameters**

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 meta-attribute of an instance. If the instance is a value, this function always returns `TRUE`. The instance must be a standalone persistent or transient object.

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Object Meta-Attributes
  An object's *meta-attributes* serve as flags that can provide information to an application, or to the object cache, about the status of an object.

## 27.4.2 OCIObjectGetProperty()

Retrieves a given property of an object.

### Purpose

Retrieves a given property of an object.

### Syntax

```
sword OCIObjectGetProperty ( OCIEnv               *envh,
                             OCIError             *errh,
                             const void           *obj,
                             OCIObjectPropId       propertyId,
                             void                 *property,
                             ub4                  *size );
```

### Parameters

**envh (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**errh (IN/OUT)**
The OCI 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 that specifies the property.

**property (OUT)**
The buffer into which the specified property is copied.

**size (IN/OUT)**
On input, this parameter specifies the size of the property buffer passed by the 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` and `OCI_OBJECTPROP_TABLE`. For non-string properties this parameter is ignored because the size is fixed.

### Comments

This function returns the specified property of the object. This 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 apply only to persistent and transient objects. An error is returned if the user tries to get a property that is not applicable to the given object. To avoid such an error, 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 communicates 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 communicates 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`

For more information about durations, see Object Duration.

**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.

**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`. The lock status of an object can also be retrieved by calling `OCIObjectIsLocked()`. Valid values include:

- `OCI_LOCK_NONE` (no lock)
- `OCI_LOCK_X` (exclusive lock)
- `OCI_LOCK_X_NOWAIT` (exclusive lock with the `NOWAIT` option)

> **See Also:**
>
> About Locking with the NOWAIT Option

**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`, 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 Topics

- OCIObjectLock()
  Locks a persistent object at the server.
- OCIObjectMarkDelete()
  Marks a standalone instance as deleted, when given a pointer to the instance.
- OCIObjectMarkUpdate()
  Marks a persistent object as updated (dirty).
- OCIObjectPin()
  Pins a referenceable object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectIsLocked()
  Gets lock status of an object.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.4.3 OCIObjectIsDirty()

Checks to see if an object is marked as dirty.

### Purpose

Checks to see if an object is marked as *dirty*.

### Syntax

```
sword OCIObjectIsDirty ( OCIEnv       *env,
                         OCIError     *err,
                         void         *ins,
                         boolean      *dirty );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 Topics**

- OCIObjectMarkUpdate()
  Marks a persistent object as updated (dirty).
- OCIObjectGetProperty()
  Retrieves a given property of an object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.4.4 OCIObjectIsLocked()

Gets lock status of an object.

### Purpose

Gets lock status of an object.

### Syntax

```
sword OCIObjectIsLocked ( OCIEnv       *env,
                          OCIError     *err,
void          *ins,
boolean       *lock );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 Topics**

- OCIObjectLock()
  Locks a persistent object at the server.
- OCIObjectGetProperty()
  Retrieves a given property of an object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.5 OCI Miscellaneous Object Functions

Lists and describes the miscellaneous object functions.

Table 27-8 describes the miscellaneous object functions that are described in this section.

**Table 27-8 Miscellaneous Object Functions**

| Function | Purpose |
|---|---|
| OCIObjectCopy() | Copy one instance to another |

**Table 27-8 (Cont.) Miscellaneous Object Functions**

| Function | Purpose |
|---|---|
| OCIObjectGetAttr() | Get an object attribute |
| OCIObjectGetInd() | Get `NULL` structure of an instance |
| OCIObjectGetObjectRef() | Return reference to a given object |
| OCIObjectGetTypeRef() | Get a reference to a TDO of an instance |
| OCIObjectLock() | Lock a persistent object |
| OCIObjectLockNoWait() | Lock a persistent object but do not wait for the lock |
| OCIObjectNew() | Create a new instance |
| OCIObjectSetAttr() | Set an object attribute |

## 27.5.1 OCIObjectCopy()

Copies a source instance to a destination.

### Purpose

Copies a source instance to a destination.

### Syntax

```
sword OCIObjectCopy ( OCIEnv              *env,
                      OCIError            *err,
                      const OCISvcCtx     *svc,
                      void                *source,
                      void                *null_source,
                      void                *target,
                      void                *null_target,
                      OCIType             *tdo,
                      OCIDuration          duration,
                      ub1                  option );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 occurring.

**source (IN)**
A pointer to the source instance; if it is an object, it must be pinned.

**See Also:**

`OCIObjectPin()`

**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, it 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 information 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, such as, 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 have been created. This can be done with `OCIObjectNew()` or `OCIObjectPin()` depending on whether the target object exists.

The `source` and `target` instances must be of the same type. If the source and target are located in different databases, then the same type must exist in both databases.

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectFlush()
  Flushes a modified persistent object to the server.
- OCIObjectNew()
  Creates a standalone instance.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.5.2 OCIObjectGetAttr()

Retrieves an object attribute.

### Purpose

Retrieves an object attribute.

### Syntax

```
sword OCIObjectGetAttr ( OCIEnv             *env,
                         OCIError           *err,
                         void               *instance,
                         void               *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,
                         void              **attr_null_struct,
                         void              **attr_value,
                         struct OCIType    **attr_tdo );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 type descriptor object (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 elements 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)**
This parameter is filled only for object and opaque attributes, not for collections. For collections (pass `OCICollGetElem`), `attr_null_struct` is `NULL`. For collections, this parameter indicates if the entire collection is `NULL` or not.

**attr_value (OUT)**
Pointer to the attribute value. See
Table 24-1 that lists the type of data and what C data type can be found in parameter `attr_value` after this call completes successfully.

**attr_tdo (OUT)**
Pointer to the TDO of the attribute.

### Comments

This function gets a value from an object. The position of the attribute returned is specified as a path expression, which is an array of names and an array of indexes. 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. The values returned are considered read-only and should not be modified in any way.

If both `attr_null_status` and `attr_null_struct` are `NULL`, no `NULL` information is returned.

### Example

For the path expression stanford.cs.stu[5].addr, the arrays appear as:

names = {"stanford", "cs", "stu", "addr"}

lengths = {8, 2, 3, 4}

indexes = {5}

### Related Topics

- OCIObjectSetAttr()
  Sets an object attribute.

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.5.3 OCIObjectGetInd()

Retrieves the `NULL` indicator structure of a standalone instance.

### Purpose

Retrieves the `NULL` indicator structure of a standalone instance.

### Syntax

```
sword OCIObjectGetInd ( OCIEnv        *env,
                        OCIError      *err,
                        void          *instance,
                        void         **null_struct );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 for a discussion of the NULL indicator structure and examples of its use

### Comments

None.

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.5.4 OCIObjectGetObjectRef()

Returns a reference to a given persistent object.

### Purpose

Returns a reference to a given persistent object.

### Syntax

```
sword OCIObjectGetObjectRef ( OCIEnv        *env,
                              OCIError      *err,
                              void          *object,
                              OCIRef        *object_ref );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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, when given a pointer to the object. Passing a value (rather than an object) to this function causes an error.

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Object Meta-Attributes
  An object's *meta-attributes* serve as flags that can provide information to an application, or to the object cache, about the status of an object.

## 27.5.5 OCIObjectGetTypeRef()

Returns a reference to the type descriptor object (TDO) of a standalone instance.

### Purpose

Returns a reference to the type descriptor object (TDO) of a standalone instance.

### Syntax

```
sword OCIObjectGetTypeRef ( OCIEnv        *env,
                            OCIError      *err,
                            void          *instance,
                            OCIRef        *type_ref );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 allocated. This can be accomplished with `OCIObjectNew()`.

### Comments

None.

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectNew()
  Creates a standalone instance.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.5.6 OCIObjectLock()

Locks a persistent object at the server.

### Purpose

Locks a persistent object at the server.

### Syntax

```
sword OCIObjectLock ( OCIEnv        *env,
                      OCIError      *err,
                      void          *object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 returns an error for transient objects and values. It also returns an error if the object does not exist.

### Related Topics

- OCIObjectPin()
  Pins a referenceable object.
- OCIObjectIsLocked()
  Gets lock status of an object.
- OCIObjectGetProperty()
  Retrieves a given property of an object.

- OCIObjectLockNoWait()
  Locks a persistent object at the server but does not wait for the lock.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- About Locking Objects for Update
  The program can optionally call `OCIObjectLock()` to lock an object for update.

## 27.5.7 OCIObjectLockNoWait()

Locks a persistent object at the server but does not wait for the lock.

### Purpose

Locks a persistent object at the server but does not wait for the lock. Returns an error if the lock is unavailable.

### Syntax

```
sword OCIObjectLockNoWait ( OCIEnv        *env,
                            OCIError      *err,
                            void          *object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 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.

`OCIObjectLockNoWait()` returns the following values:

- `OCI_INVALID_HANDLE`, if the environment handle or error handle is `NULL`
- `OCI_SUCCESS`, if the operation succeeds
- `OCI_ERROR`, if the operation fails

**Related Topics**

- OCIObjectPin()
  Pins a referenceable object.
- OCIObjectIsLocked()
  Gets lock status of an object.
- OCIObjectGetProperty()
  Retrieves a given property of an object.
- OCIObjectLock()
  Locks a persistent object at the server.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- About Locking Objects for Update
  The program can optionally call `OCIObjectLock()` to lock an object for update.

## 27.5.8 OCIObjectNew()

Creates a standalone instance.

### Purpose

Creates a standalone instance.

### Syntax

```
sword OCIObjectNew ( OCIEnv            *env,
                     OCIError          *err,
                     const OCISvcCtx   *svc,
                     OCITypeCode        typecode,
                     OCIType           *tdo,
                     void              *table,
                     OCIDuration        duration,
                     boolean            value,
                     void             **instance );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. The handle can be initialized in UTF-16 (Unicode) mode. See the description of `OCIEnvNlsCreate()`.

**err (IN/OUT)**
The OCI 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.

**typecode (IN)**
The typecode of the type of the instance.

> **See Also:**
>
> Typecodes

**tdo (IN) [optional]**
Pointer to the type descriptor object. The TDO describes the type of the instance that is to be created. See `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 that specifies a table in the server. `This` parameter can be set to `NULL` if no table is given. See the following description to learn 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. See Table 27-9 for more information.

- For a persistent object type of instance, this parameter specifies the pin duration.
- For a transient object type of instance, this parameter specifies the allocation duration and pin duration.
- For a value type of instance, 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 UTF-16 (Unicode) if the environment handle has the appropriate setting and the object is `OCIString`.

### Comments

This function creates a new instance of the type specified by the typecode or the TDO. The type can be complex or primitive.

**For Records**
When creating a package record type using `OCIObjectNew()`, clients must use typecode `OCI_TYPECODE_RECORD` when instantiating a record type.
Records are allocated in the allocation duration specified in `OCIObjectNew()`. They are subsequently freed at the end of that duration.
Record field initializers are not supported for records instantiated on the client. For instance, given the following mypack package definition, the following error is returned when you resolve it using `OCITypeByFullName()`: `OCI-22352: Type is unsupported or contains an unsupported attribute or element`. (Thus, you will never even get to call `OCIObjectNew()`).

```
create or replace package mypack is
  type r is record (rec_field number := 10);
end;
```

All records are null activated; that is, all fields of an instantiated record are set to `NULL`. In keeping with PL/SQL null semantics, all instantiated records are also atomically not-`NULL`.

**For Collections**
When creating a new instance of a package collection type, the `value` parameter must be `TRUE`. This is because package collection types cannot be persistent or referenceable, and so they must always be instantiated as values. Calling `OCIObjectNew()` for a package collection type with a `FALSE` value parameter results in an error.
Package collections are allocated in the allocation duration specified in `OCIObjectNew()`. They are subsequently freed at the end of that duration.
When creating a package, clients can use the typecodes as follows:

- `OCI_TYPECODE_NAMEDCOLLECTION` for schema level collections and package collection types
- `OCI_TYPECODE_ITABLE` for index tables
- `OCI_TYPECODE_TABLE` for nested tables
- `OCI_TYPECODE_VARRAY` for varrays

**For Booleans**
When creating new Boolean types, clients should use `OCI_TYPECODE_BOOLEAN`.

**OCI String Objects**
It can create an OCIString object with a Unicode buffer if the typecode indicates the object to be created is `OCIString`.

> **See Also:**
>
> Typecodes

Table 27-9 shows that based on the parameters `typecode` (or `tdo`), `value`, and `table`, different instances are created.

| Type of the Instance | Table != NULL | Table == NULL |
|---|---|---|
| object type (*value*=TRUE) | value | value |

| Type of the Instance | Table != NULL | Table == NULL |
|---|---|---|
| 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 an `OCIString` of 0 length. If the instance is an object, the object is marked existent but is atomically `NULL`.

> **See Also:**
>
> Create Objects Based on Object Views and Object Tables with Primary-Key-Based OIDs for information about creating new objects based on object views or user-created OIDs

**For Persistent Objects**
The object is marked dirty and existent. 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 object 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

### 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 (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 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 perform an `INSERT` or `UPDATE` operation on a FILE without first indicating a directory object and file name. Once the file name is set, the user can start reading from the FILE.

> **Note:**
>
> Oracle Database supports only binary FILEs (`BFILEs`).

**Related Topics**

- OCIObjectPinTable()
  Pins a table object for a specified duration.
- OCIObjectFree()
  Frees and unpins an object instance.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCITypeByName()
  Gets the most current version of an existing TDO.
- OCILobFileSetName()
  Sets the directory object and file name in the `BFILE` locator.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.5.9 OCIObjectSetAttr()

Sets an object attribute.

**Purpose**

Sets an object attribute.

**Syntax**

```
sword OCIObjectSetAttr ( OCIEnv            *env,
                         OCIError          *err,
                         void              *instance,
                         void              *null_struct,
                         struct OCIType    *tdo,
                         const OraText     **names,
                         const ub4         *lengths,
                         const ub4          name_count,
                         const ub4         *indexes,
                         const ub4          index_count,
                         const OCIInd       attr_null_status,
                         const void        *attr_null_struct,
                         const void        *attr_value );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 elements 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. See Table 24-1 that lists the type of data and what C data type can be found in parameter `attr_value` after this call completes successfully. Note that the given element is deep copied, and `attr_value` is strictly an input parameter.

### 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 appear as:

names = {"stanford", "cs", "stu", "addr"}

lengths = {8, 2, 3, 4}

indexes = {5}

**Related Topics**

- OCIObjectGetAttr()
  Retrieves an object attribute.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 27.6 OCI Pin, Unpin, and Free Functions

Lists and describes the OCI pin, unpin, and free functions.

Table 27-10 describes the OCI pin, unpin, and free functions that are described in this section.

**Table 27-10 Pin, Unpin, and Free Functions**

| Function | Purpose |
|---|---|
| OCICacheFree() | Free objects in the cache |
| OCICacheUnpin() | Unpin persistent objects in cache or connection |
| OCIObjectArrayPin() | Pin an array of references |
| OCIObjectFree() | Free a previously allocated object |
| OCIObjectPin() | Pin an object |
| OCIObjectPinCountReset() | Unpin an object to zero pin count |
| OCIObjectPinTable() | Pin a table object with a given duration |
| OCIObjectUnpin() | Unpin an object |

## 27.6.1 OCICacheFree()

Frees all objects and values in the cache for the specified connection.

**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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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.

### Related Topics

- OCIObjectFree()
  Frees and unpins an object instance.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> `OCIObjectFree()` for more information about freeing an instance

## 27.6.2 OCICacheUnpin()

Unpins persistent objects.

### 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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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.

**Related Topics**

- OCIObjectUnpin()
  Unpins an object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Pinning an Object
  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.
- Pin Count and Unpinning
  Each object in the object cache has a *pin count* associated with it.

## 27.6.3 OCIObjectArrayPin()

Pins an array of references.

### 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,
                          void             **obj_array,
                          ub4               *pos );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`

**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 are returned in the array. The user must allocate this array with the element type being `void *`. 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 are assigned to the elements in the array.

**Related Topics**

- OCIObjectPin()
  Pins a referenceable object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.6.4 OCIObjectFree()

Frees and unpins an object instance.

### Purpose

Frees and unpins an object instance.

### Syntax

```
sword OCIObjectFree ( OCIEnv          *env,
                      OCIError        *err,
                      void            *instance,
                      ub2              flags );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 to 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 unpinned. 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 calls `OCIObjectUnpin()` once to check if the object can be completely unpinned. 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`.

**For Values**
The memory of the object is freed immediately.

### Related Topics

- OCICacheFree()
  Frees all objects and values in the cache for the specified connection.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.6.5 OCIObjectPin()

Pins a referenceable object.

### Purpose

Pins a referenceable object.

### Syntax

```
sword OCIObjectPin ( OCIEnv             *env,
                     OCIError           *err,
                     OCIRef             *object_ref,
                     OCIComplexObject   *corhdl,
                     OCIPinOpt           pin_option,
                     OCIDuration         pin_duration,
                     OCILockOpt          lock_option,
                     void              **object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 during 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` for `OCI_DURATION_NULL`.

**lock_option (IN)**
Lock option (for example, exclusive). If a lock option is specified, the object is locked in the server. The lock status of an object can also be retrieved by calling `OCIObjectIsLocked()`. Valid values include:

- `OCI_LOCK_NONE` (no lock)
- `OCI_LOCK_X` (exclusive lock)
- `OCI_LOCK_X_NOWAIT` (exclusive lock with the `NOWAIT` option)

> **See Also:**
>
> About Locking with the NOWAIT Option

**object (OUT)**
The pointer to the pinned object.

### Comments

This function pins a referenceable object instance when given the object reference. The process of pinning serves two purposes:

- It locates an object given its reference. This is done by the object cache that keeps track of the objects in the object cache.
- It notifies the object cache that a persistent object is being used such that the persistent object cannot be aged out. Because 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 remains in memory until it is completely unpinned.

> **See Also:**
>
> `OCIObjectUnpin()`

**For Persistent Objects**
When pinning a persistent object, if it is not in the cache, the object is 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 is locked in the server if a lock option is specified.
This function returns an error for a nonexistent 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 is not 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 returns 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 Topics

- OCIObjectUnpin()
  Unpins an object.
- OCIObjectPinCountReset()
  Completely unpins an object, setting its pin count to zero.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.6.6 OCIObjectPinCountReset()

Completely unpins an object, setting its pin count to zero.

### Purpose

Completely unpins an object, setting its pin count to zero.

### Syntax

```
sword OCIObjectPinCountReset ( OCIEnv        *env,
                               OCIError      *err,
                               void          *object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 to 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. A dirty object cannot be aged out unless it is flushed.

**For Transient Objects**
The pin count of the object is decremented. A transient object can be freed only at the end of its allocation duration or when it is explicitly freed by calling `OCIObjectFree()`.

**For Values**
This function returns an error for value.

**Related Topics**

- OCIObjectPin()
  Pins a referenceable object.
- OCIObjectUnpin()
  Unpins an object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectFree()
  Frees and unpins an object instance.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- Pin Count and Unpinning
  Each object in the object cache has a *pin count* associated with it.

## 27.6.7 OCIObjectPinTable()

Pins a table object for a specified duration.

**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,
                          const OCIRef       *scope_obj_ref,
                          OCIDuration         pin_duration,
                          void              **object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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.

**scope_obj_ref (IN) [optional]**
The reference of the scoping object.

**pin_duration (IN)**
The pin duration.

> **See Also:**
>
> `OCIObjectPin()`

**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 belongs 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 Topics**

- OCIObjectPin()
  Pins a referenceable object.
- OCIObjectUnpin()
  Unpins an object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectNew()
  Creates a standalone instance.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.6.8 OCIObjectUnpin()

Unpins an object.

### Purpose

Unpins an object.

### Syntax

```
sword OCIObjectUnpin ( OCIEnv        *env,
                       OCIError      *err,
                       void          *object );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 OCI at any time without error.

This function unpins an object. An object is completely unpinned when any of the following is true:

- 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).
- It is the end of the object's pin duration.
- The function `OCIObjectPinCountReset()` is called on the object.

When an object is completely unpinned, it can be freed implicitly by OCI at any time without error.

The following rules apply to 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. A dirty object cannot be aged out unless it is flushed.

**For Transient Objects**
The pin count of the object is decremented. A transient object 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 Topics

- OCIObjectPin()
  Pins a referenceable object.
- OCIObjectPinCountReset()
  Completely unpins an object, setting its pin count to zero.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 27.7 OCI Type Information Accessor Functions

Lists and describes the OCI type information accessor functions.

Table 27-11 describes the OCI type information accessor functions that are described in this section.

**Table 27-11    Type Information Accessor Functions**

| Function | Purpose |
|---|---|
| OCITypeArrayByName() | Get an array of TDOs when given an array of object names |
| OCITypeArrayByFullName() | Get an array of TDOs when given an array of names for schema level or package level types |
| OCITypeArrayByRef() | Get an array of TDOs when given an array of object references |
| OCITypeByFullName() | Get a TDO when given a name for a schema level or package level type |
| OCITypeByName() | Get a TDO when given an object name |
| OCITypeByRef() | Get a TDO when given an object reference |
| OCITypePackage() | Get the package name of a type if it is a package type |

## 27.7.1 OCITypeArrayByName()

Gets an array of TDOs when given an array of names.

### Purpose

Gets an array of TDOs when given an array of names.

**Note:**

`OCITypeArrayByName()` does not support package level types.

### Syntax

```
sword OCITypeArrayByName ( OCIEnv           *envhp,
                           OCIError         *errhp,
                           const OCISvcCtx  *svc,
                           ub4               array_len,
                           const OraText    *schema_name[],
                           ub4               s_length[],
                           const OraText    *type_name[],
                           ub4               t_length[],
                           const OraText    *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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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` or `type_name` or `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, `schema_name` must have `array_len` number of elements. Zero (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 was available starting in release 9.0, pre-9.0 applications attempting to access an altered type generate an error. These applications must be modified, recompiled, and relinked using the latest 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. Zero (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 the current transaction) for the types retrieved. See `oro.h` for a description of each option.

**get_option (IN)**
Option for loading the types. It can be one of two values:

- `OCI_TYPEGET_HEADER` (only the header is loaded)
- `OCI_TYPEGET_ALL` (TDO and all ADO and MDOs are 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 or type name array.

You can use the `get_option` parameter 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 or 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 belongs 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 Topics

- OCITypeArrayByRef()
  Gets an array of TDOs when given an array of references.
- OCITypeByName()
  Gets the most current version of an existing TDO.
- OCITypeByRef()
  Gets a TDO when given a reference.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectGetObjectRef()
  Returns a reference to a given persistent object.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.7.2 OCITypeArrayByFullName()

Gets the most current version of an existing array of TDOs when given an array of names for schema level or package level types.

### Purpose

Gets the most current version of an existing array of TDOs when given an array of names for schema level or package level types.

> **Note:**
>
> `OCITypeArrayByFullName()` is the array version of `OCITypeByFullName()`. This means that these two functions are functionally identical and one implements `OCITypeArrayByFullName()` using `OCITypeByName()` and vice versa.

### Syntax

```
sword OCITypeArrayByFullName( OCIEnv          *env,
                              OCIError        *err,
                              const OCISvcCtx *svc,
                              ub4              array_len,
                              const oratext   *full_type_name[],
                              ub4              f_t_length[],
                              const oratext   *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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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` or `type_name` or `version_name` entries to be retrieved.

**full_type_name (IN)**
The name of the type. If the `full_type_name` is not fully qualified, name resolution will determine the type inferred. For example, `SCOTT.MYPACK.MYTYPE` would refer to type `MYTYPE` in package `MYPACK`. If the current schema is `SCOTT`, this could also be `MYPACK.MYTYPE`. See *Oracle Database PL/SQL Language Reference* for more information about PL/SQL name resolution.
This also applies for schema level types. The string could be `SCOTT.MYTYPE` or simply `MYTYPE` to specify a schema level type.

**f_t_length (IN)**
Length of `full_type_name` in bytes.

**version_name (IN)**
The version name is ignored and the latest version of the requested type is returned. Because type evolution was available starting in release 9.0, pre-9.0 applications attempting to access an altered type generate an error. These applications must be modified, recompiled, and relinked using the latest 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. Zero (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 the current transaction) for the types retrieved. See `oro.h` for a description of each option.

**get_option (IN)**
Option for loading the types. It can be one of two values:

- `OCI_TYPEGET_HEADER` (only the header is loaded)
- `OCI_TYPEGET_ALL` (TDO and all ADO and MDOs are 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 or package type name array.

You can use the `get_option` parameter 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 or package type name entry do not exist.

To retrieve a single type, rather than an array, use `OCITypeByFullName()`.

**Note:**

The TDO (array of TDOs or table definition) obtained by this function belongs 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 Topics**

- OCITypeArrayByRef()
  Gets an array of TDOs when given an array of references.
- OCITypeByFullName()
  Gets the most current version of an existing TDO when given a name for a schema level or package level type.
- OCITypeByRef()
  Gets a TDO when given a reference.
- OCITypePackage()
  Returns the package name of a type if it is a package type.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectGetObjectRef()
  Returns a reference to a given persistent object.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.7.3 OCITypeArrayByRef()

Gets an array of TDOs when given an array of references.

**Purpose**

Gets an array of TDOs when 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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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` or `type_name` or `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 the current transaction) for the types retrieved. See `oro.h` for a description of each option.

**get_option (IN)**
Option for loading the types. It can be one of two values:

- `OCI_TYPEGET_HEADER` (only the header is loaded)
- `OCI_TYPEGET_ALL` (TDO and all ADO and MDOs are 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 with the schema or 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 or 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 belongs 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 Topics**

- OCITypeArrayByName()
  Gets an array of TDOs when given an array of names.
- OCITypeByRef()
  Gets a TDO when given a reference.
- OCITypeByName()
  Gets the most current version of an existing TDO.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectGetObjectRef()
  Returns a reference to a given persistent object.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.7.4 OCITypeByFullName()

Gets the most current version of an existing TDO when given a name for a schema level or package level type.

### Purpose

Gets the most current version of an existing TDO when given a name for a schema level or package level type.

### Syntax

```
sword OCITypeByFullName(OCIEnv          *env,
                        OCIError        *err,
                        const OCISvcCtx *svc,
                        const oratext   *full_type_name,
                        ub4              f_t_length,
                        const oratext   *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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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.

**full_type_name (IN)**
The name of the type. If the `full_type_name` is not fully qualified, name resolution will determine the type inferred. Thus, the full path name to the object can include the schema, package name, and type name. For example, `SCOTT.MYPACK.MYTYPE` would refer to type `MYTYPE` in package `MYPACK`. If the current schema is `SCOTT`, this could also be `MYPACK.MYTYPE`. See *Oracle Database PL/SQL Language Reference* for more information about PL/SQL name resolution.
This also applies for schema level types. The string could be `SCOTT.MYTYPE` or simply `MYTYPE` to specify a schema level type.

**f_t_length (IN)**
Length of `full_type_name` in bytes.

**version_name (IN)**
The version name is ignored and the latest version of the requested type is returned. Because type evolution was available starting in release 9.0, pre-9.0 applications attempting to access an altered type generate an error. These applications must be modified, recompiled, and relinked using the latest 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

**get_option ((IN)**
Option for loading the types. It can be one of two values:

- `OCI_TYPEGET_HEADER` (only the header is loaded)
- `OCI_TYPEGET_ALL` (TDO and all ADO and MDOs are loaded)

**tdo (OUT)**
Pointer to the pinned type in the object cache.

### Comments

The fact that the type name is passed as a single string also enables other clients and drivers, such as thick-client JDBC, to easily resolve type names contained in a single name string.

Package types which contain remote package type fields will not be supported by `OCITypeByFullName()`. Any attempt to get a package type, which contains a remote package type field, results in an error.

This function gets a pointer to the existing type associated with the schema or package type name. It returns an error if any of the required parameters is `NULL`, or if the object type associated with the schema or package type name does not exist, or if `version_name` does not exist.

> **Note:**
>
> Schema and package type names are case-sensitive. If they have been created with SQL, you must use strings in all uppercase, or the program stops.

This function always makes a round-trip to the server. Therefore calling this function repeatedly to get the type can significantly reduce performance. To minimize the round-trips, the application can call the function for each type and cache the type objects.

To free the type obtained by this function, call `OCIObjectUnpin()` or `OCIObjectPinCountReset()`.

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 belongs 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 Topics**

- OCITypeByRef()
  Gets a TDO when given a reference.
- OCITypeArrayByFullName()
  Gets the most current version of an existing array of TDOs when given an array of names for schema level or package level types.
- OCITypeArrayByRef()
  Gets an array of TDOs when given an array of references.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectUnpin()
  Unpins an object.

- OCIObjectPinCountReset()
  Completely unpins an object, setting its pin count to zero.
- OCITypeArrayByName()
  Gets an array of TDOs when given an array of names.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.7.5 OCITypeByName()

Gets the most current version of an existing TDO.

### Purpose

Gets the most current version of an existing TDO. This call does not support package level types. The name of the schema and the name of the type are each entered in separate strings.

### Syntax

```
sword OCITypeByName ( OCIEnv              *env,
                      OCIError            *err,
                      const OCISvcCtx     *svc,
                      const OraText       *schema_name,
                      ub4                  s_length,
                      const OraText       *type_name,
                      ub4                  t_length,
                      const OraText       *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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 uppercase, or 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 uppercase, or 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 was available starting in release 9.0, pre-9.0 applications attempting to access an altered type generate an error. These applications must be modified, recompiled, and relinked using the latest 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

**get_option ((IN)**
Option for loading the types. It can be one of two values:

- `OCI_TYPEGET_HEADER` (only the header is loaded)
- `OCI_TYPEGET_ALL` (TDO and all ADO and MDOs are loaded)

**tdo (OUT)**
Pointer to the pinned type in the object cache.

### Comments

This function gets a pointer to the existing type associated with the schema or type name. It returns an error if any of the required parameters is `NULL`, or if the object type associated with the schema or 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 must use strings in all uppercase, or the program stops.

This function always makes a round-trip to the server. Therefore calling this function repeatedly to get the type can significantly reduce performance. To minimize the round-trips, the application can call the function for each type and cache the type objects.

To free the type obtained by this function, call `OCIObjectUnpin()` or `OCIObjectPinCountReset()`.

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 belongs 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 Topics**

- OCITypeByRef()
  Gets a TDO when given a reference.
- OCITypeArrayByName()
  Gets an array of TDOs when given an array of names.
- OCITypeArrayByRef()
  Gets an array of TDOs when given an array of references.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIObjectUnpin()
  Unpins an object.
- OCIObjectPinCountReset()
  Completely unpins an object, setting its pin count to zero.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.7.6 OCITypeByRef()

Gets a TDO when given a reference.

**Purpose**

Gets a TDO when 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 descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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 the current transaction for the type to retrieve. See `oro.h` for a description of each option.

**get_option (IN)**
Option for loading the type. It can be one of two values:

- `OCI_TYPEGET_HEADER` (only the header is loaded)
- `OCI_TYPEGET_ALL` (TDO and all ADO and MDOs are 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 belongs 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 Topics

- OCITypeByName()
  Gets the most current version of an existing TDO.
- OCITypeArrayByName()
  Gets an array of TDOs when given an array of names.
- OCITypeArrayByRef()
  Gets an array of TDOs when given an array of references.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 27.7.7 OCITypePackage()

Returns the package name of a type if it is a package type.

### Purpose

Returns the package name of a type if it is a package type.

### Syntax

```
oratext* OCITypePackage( OCIEnv        *env,
                         OCIError      *err,
                         const OCIType *tdo,
                         ub4           *n_length);
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**tdo (OUT)**
Pointer to the pinned type in the object cache.

**n_length**
Length of the package name, in bytes.

### Comments

If the type is not a package type, the return value will be null and `n_length` will be zero.

### Related Topics

- OCITypeArrayByFullName()
  Gets the most current version of an existing array of TDOs when given an array of names for schema level or package level types.
- OCITypeByFullName()
  Gets the most current version of an existing TDO when given a name for a schema level or package level type.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28

# OCI Data Type Mapping and Manipulation Functions

This chapter describes the OCI data type mapping and manipulation functions.

These functions are Oracle's external C language interface to Oracle Database predefined types.

> **See Also:**
>
> For code examples, see the demonstration programs included with your Oracle Database installation. For additional information, see OCI Demonstration Programs.

This chapter contains these topics:

- Introduction to Data Type Mapping and Manipulation Functions
- OCI Collection and Iterator Functions
- OCI Date, Datetime, and Interval Functions
- OCI NUMBER Functions
- OCI Raw Functions
- OCI REF Functions
- OCI String Functions
- OCI Table Functions

## 28.1 Introduction to Data Type Mapping and Manipulation Functions

Introduces detailed information about OCI data type mapping and manipulation functions.

This section describes the OCI data type mapping and manipulation functions in detail.

> **See Also:**
>
> Object-Relational Data Types in OCI for more information about the functions listed in this chapter

## 28.1.1 Conventions for OCI Functions

Reports the conventions used for describing OCI functions.

See the Conventions for OCI Functions for the conventions used in describing each function. The entries for each function may also contain the following information:

## 28.1.2 Returns

Describes the value returned by the function.

A description of what value is returned by the function if the function returns something other than the standard return codes listed in "Table 28-1".

## 28.1.3 Data Type Mapping and Manipulation Function Return Values

Describes the function return values.

The OCI data type mapping and manipulation functions typically return one of the values shown in Table 28-1.

**Table 28-1 Function Return Values**

| Return Value | Meaning |
|---|---|
| `OCI_SUCCESS` | The operation succeeded. |
| `OCI_ERROR` | The operation failed. The specific error can be retrieved by calling `OCIErrorGet()` on the error handle passed to the function. |
| `OCI_INVALID_HANDLE` | The OCI handle passed to the function is invalid. |

**See Also:**

Error Handling in OCI for more information about return codes and error handling

## 28.1.4 Functions Returning Other Values

Lists functions returning values directly from the function call.

Some functions return values other than those listed in Table 28-1. When you use the following functions, consider that they return a value directly from the function call, rather than through an OUT parameter.

- OCICollMax()
- OCIRawPtr()
- OCIRawSize()
- OCIRefHexSize()

- OCIRefIsEqual()
- OCIRefIsNull()
- OCIStringPtr()
- OCIStringSize()

### 28.1.5 Server Round-Trips for Data Type Mapping and Manipulation Functions

Indicates the number of server round-trips required for individual OCI data type mapping and manipulation functions.

For a table showing the number of server round-trips required for individual OCI data type mapping and manipulation functions, see Table C-5.

### 28.1.6 Examples

Describes information about the examples.

For more information about these functions, including some code examples, see Object-Relational Data Types in OCI.

## 28.2 OCI Collection and Iterator Functions

Lists and describes the OCI collection and iterator functions.

Table 28-2 describes the OCI collection and iterator functions that are described in this section.

**Table 28-2    Collection and Iterator Functions**

| Function | Purpose |
|---|---|
| OCICollAppend() | Append an element to the end of a collection |
| OCICollAssign() | Assign (deep copy) one collection to another |
| OCICollAssignElem() | Assign the given element value `elem` to the element at `coll[index]` |
| OCICollGetElem() | Get pointer to an element |
| OCICollGetElemArray() | Get an array of elements from a collection |
| OCICollIsLocator() | Indicate whether a collection is locator-based or not |
| OCICollMax() | Return maximum number of elements in collection |
| OCICollSize() | Get current size of collection (in number of elements) |
| OCICollTrim() | Trim elements from the collection |
| OCIIterCreate() | Create iterator to scan the varray elements |
| OCIIterDelete() | Delete iterator |
| OCIIterGetCurrent() | Get current collection element |

**Table 28-2 (Cont.) Collection and Iterator Functions**

| Function | Purpose |
|---|---|
| OCIIterInit() | Initialize iterator to scan the given collection |
| OCIIterNext() | Get next collection element |
| OCIIterPrev() | Get previous collection element |

## 28.2.1 OCICollAppend()

Appends an element to the end of a collection.

### Purpose

Appends an element to the end of a collection.

### Syntax

```
sword OCICollAppend ( OCIEnv            *env,
                      OCIError          *err,
                      const void        *elem,
                      const void        *elemind,
                      OCIColl           *coll );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**elem (IN)**
Pointer to the element to be 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 is set to non-`NULL`.

**coll (IN/OUT)**
Updated collection.

### Comments

Appending an element is equivalent to increasing the size of the collection by one 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.

`OCICollAppend()` creates an element at index 0 if no elements exist. The method (array bind) for binding PL/SQL index-by tables assumes a start index of 1.

**Returns**

This function returns an error if the current size of the collection equals the maximum size (upper bound) of the collection before appending the element. This function also returns an error if any of the input parameters is `NULL`.

**Related Topics**

- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.2 OCICollAssign()

Assigns (deep copies) one collection to another.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**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` collection contains any elements, then the elements are deleted before the assignment. This function performs a deep copy. The memory for the elements comes from the object cache.

**Returns**

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 the `lhs` collection is less than the current number of elements in the `rhs` collection

**Related Topics**

- OCICollAssignElem()
  Assigns the given element value `elem` to the element at `coll[index]`.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.3 OCICollAssignElem()

Assigns the given element value `elem` to the element at `coll[index]`.

**Purpose**

Assigns the given element value `elem` to the element at `coll[index]`.

**Syntax**

```
sword OCICollAssignElem ( OCIEnv          *env,
                          OCIError        *err,
                          sb4              index,
                          const void      *elem,
                          const void      *elemind,
                          OCIColl         *coll );
```

**Parameters**

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**index (IN)**
Index of the element to which the element is assigned.

**elem (IN)**
Source element from which the element is assigned. See Table 24-1 that lists the type of data and what C data type can be found in parameter `elem`.

**elemind (IN) [optional]**
Pointer to the element's `NULL` indicator information; if (`elemind == NULL`), then the `NULL` indicator information of the assigned element is 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 might not exist, as 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.

### Returns

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 Topics**

- OCICollAssign()
  Assigns (deep copies) one collection to another.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.4 OCICollGetElem()

Gets a pointer to the element at the given index.

### 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,
                       void            **elem,
                       void            **elemind );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**coll (IN)**
Pointer to the element in this collection to be 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 element to be returned.

**elemind (OUT) [optional]**
Address of the `NULL` indicator information is returned. If (`elemind` == `NULL`), then the `NULL` indicator information is not 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.

Table 28-3 describes for each OCI type code value, the element pointer type, and what the corresponding collection element type is. The element pointer is returned with the `elem` parameter of `OCICollGetElem()`. See the table for object data type mappings for object type attributes in About Mapping Object Data Types to Cfor a more complete list of the `OCITypeCode` values.

**Table 28-3 Element Pointers**

| OCITypeCode Values | *elem Is Set to | Element Type |
|---|---|---|
| OCI_TYPECODE_NUMBER | `OCINumber*` | Oracle `NUMBER` (`OCINumber`) |
| OCI_TYPECODE_DATE | `OCIDate*` | Date (`OCIDate`) |

**Table 28-3 (Cont.) Element Pointers**

| OCITypeCode Values | *elem Is Set to | Element Type |
|---|---|---|
| OCI_TYPECODE_TIMESTAMP | `OCIDateTime*` | Datetime (`OCIDateTime`) |
| OCI_TYPECODE_INTERVAL_YM,<br>OCI_TYPECODE_INTERVAL_DS | `OCIInterval*` | Interval (`OCIInterval`) |
| OCI_TYPECODE_CHAR (n),<br>OCI_TYPECODE_NCHAR | `OCIString**` | Variable-length string (`OCIString*`) |
| OCI_TYPECODE_RAW | `OCIRaw**` | Variable-length raw (`OCIRaw*`) |
| OCI_TYPECODE_REF | `OCIRef**` | Object reference (`OCIRef*`) |
| OCI_TYPECODE_BLOB,<br>OCI_TYPECODE_CLOB,<br>OCI_TYPECODE_NCLOB | `OCILobLocator**` | Lob locator (`OCILobLocator*`) |
| OCI_TYPECODE_OBJECT | `person*` | Object type (such as person) |
| OCI_TYPECODE_NAMEDCOLLECTION | `OCIColl *` | Collections |

The element pointer returned by `OCICollGetElem()` is in a form that can be used not only to access the element data but also to serve 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 the pointer to a reference handle (`OCIRef**`). After getting the pointer to the collection element, you may want to modify it by assigning a new reference.

The following code example shows how this can be accomplished with the `OCIRefAssign()` function.

Assigning a New Reference to the Pointer to the Collection Element

```
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` is allocated by `OCIRefAssign()` and returned in the `target` parameter.

Similarly, if the collection element was of type string (`OCIString*`), `OCICollGetElem()` returns the pointer to the 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 following code example shows the prototype of the `OCINumberAssign()` call.

Prototype of OCINumberAssign() Call

```
sword OCINumberAssign(OCIError        *err,
                      const OCINumber *from,
                      OCINumber       *to);
```

**Returns**

The `OCICollGetElem()` function returns an error if any of the input parameters is `NULL`.

**Related Topics**

- OCICollAssignElem()
  Assigns the given element value `elem` to the element at `coll[index]`.
- OCIRefAssign()
  Assigns one `REF` to another, such that both reference the same object.
- OCIStringAssign()
  Assigns one string to another string.
- OCIStringAssignText()
  Assigns the source text string to the target string.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.5 OCICollGetElemArray()

Gets an array of elements from a collection when given a starting index.

**Purpose**

Gets an array of elements from a collection when given a starting index.

**Syntax**

```
sword OCICollGetElemArray ( OCIEnv             *env,
                            OCIError           *err,
                            const OCIColl      *coll,
                            sb4                 index,
                            boolean            *exists,
                            void              **elem,
                            void              **elemind,
                            uword              *nelems );
```

**Parameters**

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**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; otherwise, it is set to `TRUE`.

**elem (OUT)**
Address of the desired elements to be returned.

**elemind (OUT) [optional]**
Address of the `NULL` indicator information to be returned. If (`elemind == NULL`), then the `NULL` indicator information is *not* returned.

**nelems (IN)**
Maximum number of pointers to both `elem` and `elemind`.

### Comments

Gets the address of the elements from the given position.

For index by integer collections, `OCICollGetElemArray()` gets the elements beginning at the given index, but loses the index information for each element in the process. Users are able to get the element data beginning at that index as an array, but cannot get the index for each element in the array. This behavior is similar to what happens for the `OCIIterCreate()`, `OCIIterDelete()`, `OCIIterGetCurrent()`, `OCIIterInit()`, `OCIIterNext()`, and `OCIIterPrev()` functions.

### Returns

Optionally, this function also returns the address of the element's `NULL` indicator information.

### Related Topics

- OCICollGetElem()
  Gets a pointer to the element at the given index.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIIterCreate()
  Creates an iterator to scan the elements or the collection.
- OCIIterDelete()
  Deletes a collection iterator.

- OCIIterGetCurrent()
  Gets a pointer to the current iterator collection element.
- OCIIterInit()
  Initializes an iterator to scan a collection.
- OCIIterNext()
  Gets a pointer to the next iterator collection element.
- OCIIterPrev()
  Gets a pointer to the previous iterator collection element.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.6 OCICollIsLocator()

Indicates whether a collection is locator-based or not.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**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 a collection is locator-based.

### Returns

Returns `TRUE` in the `result` parameter if the collection item is locator-based; otherwise, it returns `FALSE`.

### Related Topics

- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.7 OCICollMax()

Gets the maximum size in number of elements of the given collection.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**coll (IN)**
Collection whose number of elements is returned. The `coll` parameter 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.

For collections that do not support negative indexes, the largest index number is also the maximum size of the collection. However, this is not true for index-by integer collections because some of the elements can have negative indexes, so the largest index numbered element is not the same as the maximum collection size.

### Returns

The upper bound of the given collection.

The return value is always 0 (no upper bound) for index-by integer collections.

### Related Topics

- OCICollSize()
  Gets the current size in number of elements of the given collection.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.8 OCICollSize()

Gets the current size in number of elements of the given collection.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**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 a nested table, this count is not decremented when elements are deleted. So, this count includes any *holes* created by deleting elements. A trim operation (`OCICollTrim()`) decrements 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
```

### Returns

The `OCICollSize()` function returns an error if an error occurs during the loading of the collection into the object cache or if any of the input parameters is `NULL`.

### Related Topics

- OCICollMax()
  Gets the maximum size in number of elements of the given collection.
- OCICollTrim()
  Trims the given number of elements from the end of the collection.
- OCITableSize()
  Returns the size of the given table, not including any holes created by deleted elements.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.9 OCICollTrim()

Trims the given number of elements from the end of the collection.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**trim_num (IN)**
Number of elements to trim.

**coll (IN/OUT)**
Removes (frees) `trim_num` of elements from the end of the collection `coll`.

### Comments

The elements are removed from the end of the collection.

### Returns

An error is returned if `trim_num` is greater than the current size of the collection.

**Related Topics**

- OCICollSize()
  Gets the current size in number of elements of the given collection.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.10 OCIIterCreate()

Creates an iterator to scan the elements or the collection.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**coll (IN)**
Collection that is scanned. For Oracle8*i* or later, valid collection types include varrays and nested tables.

**itr (OUT)**
Address to the allocated collection iterator to be 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 an "at beginning of collection" error is returned.

For index-by integer collections, the `OCIIterCreate()`, `OCIIterDelete()`, `OCIIterGetCurrent()`, `OCIIterInit()`, `OCIIterNext()`, and `OCIIterPrev()` functions all ignore the index for each element in the collection. That is, `OCIIterGetCurrent()` returns only the element value and not the index of the element.

### Returns

This function returns an error if any of the input parameters is `NULL`.

### Related Topics

- OCIIterDelete()
  Deletes a collection iterator.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIIterGetCurrent()
  Gets a pointer to the current iterator collection element.
- OCIIterInit()
  Initializes an iterator to scan a collection.
- OCIIterNext()
  Gets a pointer to the next iterator collection element.

- OCIIterPrev()
  Gets a pointer to the previous iterator collection element.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.11 OCIIterDelete()

Deletes a collection iterator.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**itr (IN/OUT)**
The allocated collection iterator that is destroyed and set to `NULL` before returning.

### Comments

Deletes an iterator that was previously created by a call to `OCIIterCreate()`.

For index-by integer collections, the `OCIIterCreate()`, `OCIIterDelete()`, `OCIIterGetCurrent()`, `OCIIterInit()`, `OCIIterNext()`, and `OCIIterPrev()` functions all ignore the index for each element in the collection. That is, `OCIIterGetCurrent()` returns only the element value and not the index of the element.

### Returns

This function returns an error if any of the input parameters is `NULL`.

### Related Topics

- OCIIterCreate()
  Creates an iterator to scan the elements or the collection.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIIterGetCurrent()
  Gets a pointer to the current iterator collection element.
- OCIIterInit()
  Initializes an iterator to scan a collection.
- OCIIterNext()
  Gets a pointer to the next iterator collection element.
- OCIIterPrev()
  Gets a pointer to the previous iterator collection element.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.12 OCIIterGetCurrent()

Gets a pointer to the current iterator collection element.

### Purpose

Gets a pointer to the current iterator collection element.

### Syntax

```
sword OCIIterGetCurrent ( OCIEnv            *env,
                          OCIError          *err,
                          const OCIIter     *itr,
                          void             **elem,
                          void             **elemind );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**itr (IN)**
Iterator that points to the current element.

**elem (OUT)**
Address of the element pointed to by the iterator to be returned. See Table 24-1 that lists the type of data and what C data type can be found in parameter `elem`.

**elemind (OUT) [optional]**
Address of the element's `NULL` indicator information to be returned; if (`elem_ind` == NULL) then the `NULL` indicator information is *not* returned.

### Comments

Returns the pointer to the current iterator collection element and its corresponding `NULL` information.

For index-by integer collections, the `OCIIterCreate()`, `OCIIterGetCurrent()`, `OCIIterInit()`, `OCIIterNext()`, and `OCIIterPrev()` functions all ignore the index for each element in the collection. That is, `OCIIterGetCurrent()` returns only the element value and not the index of the element.

### Returns

This function returns an error if any input parameter is `NULL`.

### Related Topics

- OCIIterCreate()
  Creates an iterator to scan the elements or the collection.
- OCIIterDelete()
  Deletes a collection iterator.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIIterInit()
  Initializes an iterator to scan a collection.
- OCIIterNext()
  Gets a pointer to the next iterator collection element.
- OCIIterPrev()
  Gets a pointer to the previous iterator collection element.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.13 OCIIterInit()

Initializes an iterator to scan a collection.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**coll (IN)**
Collection that is 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 at the given iterator to point to the beginning of the given collection. You can use this function to perform either of the following tasks:

- Reset an iterator to point back to the beginning of the collection.
- Reuse an allocated iterator to scan a different collection.

For index-by integer collections, the `OCIIterCreate()`, `OCIIterDelete()`, `OCIIterGetCurrent()`, `OCIIterInit()`, `OCIIterNext()`, and `OCIIterPrev()` functions all ignore the index for each element in the collection. That is, `OCIIterGetCurrent()` returns only the element value and not the index of the element.

### Returns

Returns an error if any input parameter is `NULL`.

**Related Topics**

- OCIIterCreate()
  Creates an iterator to scan the elements or the collection.
- OCIIterDelete()
  Deletes a collection iterator.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIIterGetCurrent()
  Gets a pointer to the current iterator collection element.
- OCIIterNext()
  Gets a pointer to the next iterator collection element.

- OCIIterPrev()
  Gets a pointer to the previous iterator collection element.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.14 OCIIterNext()

Gets a pointer to the next iterator collection element.

### Purpose

Gets a pointer to the next iterator collection element.

### Syntax

```
sword OCIIterNext ( OCIEnv            *env,
                    OCIError          *err,
                    OCIIter           *itr,
                    void             **elem,
                    void             **elemind,
                    boolean           *eoc);
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`..

**itr (IN/OUT)**
Iterator that is updated to point to the next element.

**elem (OUT)**
Address of the next element; returned after the iterator is updated to point to it. See Table 24-1 that lists the type of data and what C data type can be found in parameter `elem`.

**elemind (OUT) [optional]**
Address of the element's `NULL` indicator information; if (`elem_ind` == `NULL`), then the `NULL` indicator information is *not* returned.

**eoc (OUT)**
`TRUE` if the iterator is at the end of the collection (that is, the 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 before you execute this function, then calling this function sets the `eoc` flag to `TRUE`. The iterator is left unchanged in that case.

For index-by integer collections, the `OCIIterCreate()`, `OCIIterDelete()`, `OCIIterGetCurrent()`, `OCIIterInit()`, `OCIIterNext()`, and `OCIIterPrev()` functions all ignore the index for each element in the collection. That is, `OCIIterGetCurrent()` returns only the element value and not the index of the element.

### Returns

This function returns an error if any input parameter is `NULL`.

**Related Topics**

- OCIIterCreate()
  Creates an iterator to scan the elements or the collection.
- OCIIterDelete()
  Deletes a collection iterator.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIIterGetCurrent()
  Gets a pointer to the current iterator collection element.
- OCIIterInit()
  Initializes an iterator to scan a collection.
- OCIIterPrev()
  Gets a pointer to the previous iterator collection element.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.2.15 OCIIterPrev()

Gets a pointer to the previous iterator collection element.

### Purpose

Gets a pointer to the previous iterator collection element.

### Syntax

```
sword OCIIterPrev ( OCIEnv           *env,
                    OCIError         *err,
                    OCIIter          *itr,
                    void            **elem,
                    void            **elemind,
                    boolean          *boc );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**itr (IN/OUT)**
Iterator that 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. See Table 24-1 that lists the type of data and what C data type can be found in parameter `elem`.

**elemind (OUT) [optional]**
Address of the element's `NULL` indicator information; if (`elemind == NULL`), then the `NULL` indicator information is *not* returned.

**boc (OUT)**
`TRUE` if iterator is at the beginning of the collection (that is, the 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 before you execute this function, then calling this function sets `boc` to `TRUE`. The iterator is left unchanged in that case.

For index-by integer collections, the `OCIIterCreate()`, `OCIIterDelete()`, `OCIIterGetCurrent()`, `OCIIterInit()`, `OCIIterNext()`, and `OCIIterPrev()` functions all ignore the index for each element in the collection. That is, `OCIIterGetCurrent()` returns only the element value and not the index of the element.

### Returns

This function returns an error if any input parameter is `NULL`.

### Related Topics

- OCIIterCreate()
  Creates an iterator to scan the elements or the collection.
- OCIIterDelete()
  Deletes a collection iterator.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIIterGetCurrent()
  Gets a pointer to the current iterator collection element.
- OCIIterInit()
  Initializes an iterator to scan a collection.
- OCIIterNext()
  Gets a pointer to the next iterator collection element.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3 OCI Date, Datetime, and Interval Functions

Lists and describes the OCI date and interval functions.

Table 28-4 describes the OCI date and interval functions that are described in this section.

**Table 28-4 Date Functions**

| Function | Purpose |
|---|---|
| OCIDateAddDays() | Add or subtract days |
| OCIDateAddMonths() | Add or subtract months |
| OCIDateAssign() | Assign date |
| OCIDateCheck() | Check if the given date is valid |
| OCIDateCompare() | Compare dates |
| OCIDateDaysBetween() | Get number of days between two dates |
| OCIDateFromText() | Convert string to date |
| OCIDateGetDate() | Get the date portion of a date |
| OCIDateGetTime() | Get the time portion of a date |
| OCIDateLastDay() | Get date of last day of month |
| OCIDateNextDay() | Get date of next day |
| OCIDateSetDate() | Set the date portion of a date |
| OCIDateSetTime() | Set the time portion of a date |
| OCIDateSysDate() | Get the current system date and time |
| OCIDateTimeAssign() | Perform a datetime assignment |
| OCIDateTimeCheck() | Check if the given date is valid |
| OCIDateTimeCompare() | Compare two datetime values |
| OCIDateTimeConstruct() | Construct a datetime descriptor |

**Table 28-4 (Cont.) Date Functions**

| Function | Purpose |
|---|---|
| OCIDateTimeConvert() | Convert one datetime type to another |
| OCIDateTimeFromArray() | Convert an array of size `OCI_DT_ARRAYLEN` to an `OCIDateTime` descriptor |
| OCIDateTimeFromText() | Convert the given string to Oracle datetime type in the `OCIDateTime` descriptor, according to the specified format |
| OCIDateTimeGetDate() | Get the date (year, month, day) portion of a datetime value |
| OCIDateTimeGetTime() | Get the time (hour, min, second, fractional second) of a datetime value |
| OCIDateTimeGetTimeZoneName() | Get the time zone name portion of a datetime value |
| OCIDateTimeGetTimeZoneOffset() | Get the time zone (hour, minute) portion of a datetime value |
| OCIDateTimeIntervalAdd() | Add an interval to a datetime to produce a resulting datetime |
| OCIDateTimeIntervalSub() | Subtract an interval from a datetime and store the result in a datetime |
| OCIDateTimeSubtract() | Take two datetimes as input and store their difference in an interval |
| OCIDateTimeSysTimeStamp() | Get the system current date and time as a time stamp with time zone |
| OCIDateTimeToArray() | Convert an `OCIDateTime` descriptor to an array |
| OCIDateTimeToText() | Convert the given date to a string according to the specified format |
| OCIDateToText() | Convert date to string |
| OCIDateZoneToZone() | Convert date from one time zone to another zone |
| OCIIntervalAdd() | Add two intervals to produce a resulting interval |
| OCIIntervalAssign() | Copy one interval to another |
| OCIIntervalCheck() | Check the validity of an interval |
| OCIIntervalCompare() | Compare two intervals |
| OCIIntervalDivide() | Divide an interval by an Oracle `NUMBER` to produce an interval |
| OCIIntervalFromNumber() | Convert an Oracle `NUMBER` to an interval |
| OCIIntervalFromText() | When given an interval string, return the interval represented by the string |

**Table 28-4 (Cont.) Date Functions**

| Function | Purpose |
|---|---|
| OCIIntervalFromTZ() | Return an `OCI_DTYPE_INTERVAL_DS` |
| OCIIntervalGetDaySecond() | Get values of day, hour, minute, and second from an interval |
| OCIIntervalGetYearMonth() | Get year and month from an interval |
| OCIIntervalMultiply() | Multiply an interval by an Oracle `NUMBER` to produce an interval |
| OCIIntervalSetDaySecond() | Set day, hour, minute, and second in an interval |
| OCIIntervalSetYearMonth() | Set year and month in an interval |
| OCIIntervalSubtract() | Subtract two intervals and stores the result in an interval |
| OCIIntervalToNumber() | Convert an interval to an Oracle NUMBER |
| OCIIntervalToText() | When given an interval, produce a string representing the interval |

## 28.3.1 OCIDateAddDays()

Adds or subtracts days from a given date.

### 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)**
he 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`.

### Returns

This function returns an error if an invalid date is passed to it.

### Related Topics

- OCIDateAddMonths()
  Adds or subtracts months from a given date.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.2 OCIDateAddMonths()

Adds or subtracts months from a given date.

### 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`.

### Returns

This function returns an error if an invalid date is passed to it.

### Related Topics

- OCIDateAddDays()
  Adds or subtracts days from a given date.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.3 OCIDateAssign()

Performs a date assignment.

### 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 Topics**

- OCIDateCheck()
  Checks if the given date is valid.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.4 OCIDateCheck()

Checks if the given date is valid.

### 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, it returns the logical operator OR combination of all error bits specified in Table 28-5.

**Table 28-5 Error Bits Returned by the valid Parameter for OCIDateCheck()**

| 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.
```

### Returns

This function returns an error if `date` or `valid` pointer is `NULL`.

### Related Topics

- OCIDateCompare()
  Compares two dates.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.5 OCIDateCompare()

Compares two dates.

### 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 as listed in Table 28-6.

**Table 28-6 Comparison Results**

| Comparison Result | Output in result Parameter |
|---|---|
| `date1 < date2` | -1 |
| `date1 = date2` | 0 |
| `date1 > date2` | 1 |

### Returns

This function returns an error if an invalid date is passed to it.

### Related Topics

- OCIDateCheck()
  Checks if the given date is valid.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.6 OCIDateDaysBetween()

Gets the number of days between two dates.

### 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.

### Returns

This function returns an error if an invalid date is passed to it.

### Related Topics

- OCIDateCheck()
  Checks if the given date is valid.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.7 OCIDateFromText()

Converts a character string to a date type according to the specified format.

### Purpose

Converts a character string to a date type according to the specified format.

### Syntax

```
sword OCIDateFromText ( OCIError          *err,
                        const OraText     *date_str,
                        ub4                d_str_length,
                        const OraText     *fmt,
                        ub1                fmt_length,
                        const OraText     *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

See the `TO_DATE` conversion function described in the *Oracle Database SQL Language Reference*for a description of format and multilingual arguments.

### Returns

This function returns an error if it receives an invalid format, language, or input string.

**Related Topics**

- OCIDateToText()
  Converts a date type to a character string.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.8 OCIDateGetDate()

Gets the year, month, and day stored in an Oracle date.

### Purpose

Gets 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 Topics**

- OCIDateSetDate()
  Set the values in an Oracle date.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.9 OCIDateGetTime()

Gets the time stored in an Oracle date.

### 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.

#### Returns

Returns the time information in the form: hour, minute and seconds.

#### Related Topics

- OCIDateSetTime()
  Sets the time information in an Oracle date.
- OCIDateGetDate()
  Gets the year, month, and day stored in an Oracle date.

## 28.3.10 OCIDateLastDay()

Gets the date of the last day of the month in a specified date.

#### 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`.

#### Returns

This function returns an error if an invalid date is passed to it.

#### Related Topics

- OCIDateGetDate()
  Gets the year, month, and day stored in an Oracle date.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.11 OCIDateNextDay()

Gets the date of the next day of the week after a given date.

### Purpose

Gets the date of the 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 the 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` that is later than `date`.

### Returns

Returns the date of the first day of the week named by `day` that is later than `date`.

### Example

The following code example shows how to get the date of the next Monday after April 18, 1996 (a Thursday).

Getting the Date for a Specific Day After a Specified Date

```
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 an error if an invalid date or day is passed to it.

### Related Topics

- OCIDateGetDate()
  Gets the year, month, and day stored in an Oracle date.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.12 OCIDateSetDate()

Set the values in an Oracle date.

### 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 Topics

- OCIDateGetDate()
  Gets the year, month, and day stored in an Oracle date.

## 28.3.13 OCIDateSetTime()

Sets the time information in an Oracle date.

### 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 Topics

- OCIDateGetTime()
  Gets the time stored in an Oracle date.

## 28.3.14 OCIDateSysDate()

Gets the current system date and time of the client.

### 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 Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.15 OCIDateTimeAssign()

Performs a datetime assignment.

### Purpose

Performs a datetime assignment.

### Syntax

```
sword OCIDateTimeAssign ( void                 *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 occurs 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, right-hand side (`rhs`) datetime to be assigned.

**to (OUT)**
Target, left-hand side (`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`; or OCI_ERROR.

### Related Topics

- OCIDateTimeCheck()
  Checks if the given date is valid.
- OCIDateTimeConstruct()
  Constructs a datetime descriptor.

## 28.3.16 OCIDateTimeCheck()

Checks if the given date is valid.

### Purpose

Checks if the given date is valid.

### Syntax

```
sword OCIDateTimeCheck ( void                 *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 occurs 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 logical operator OR combination of all the error bits specified in Table 28-7.

**Table 28-7 Error Bits Returned by the valid Parameter for OCIDateTimeCheck()**

| 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) |

**Table 28-7 (Cont.) Error Bits Returned by the valid Parameter for OCIDateTimeCheck()**

| Macro Name | Bit Number | Error |
|---|---|---|
| `OCI_DT_DAY_MISSING_FROM_1582` | 0x1000 | Day is one of those missing from 1582 |
| `OCI_DT_YEAR_ZERO` | 0x2000 | Year may not equal zero |
| `OCI_DT_INVALID_TIMEZONE` | 0x4000 | Bad time zone |
| `OCI_DT_INVALID_FORMAT` | 0x8000 | Bad date format input |

So, for example, if the date passed in was 2/0/1990 25:61:10 in (month/day/year hours:minutes:seconds format), the error returned 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 Topics

- OCIDateTimeAssign()
  Performs a datetime assignment.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.17 OCIDateTimeCompare()

Compares two datetime values.

### Purpose

Compares two datetime values.

### Syntax

```
sword OCIDateTimeCompare ( void                *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 as listed in Table 28-8.

**Table 28-8 Comparison Results Returned by the result Parameter for OCIDateTimeCompare()**

| 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 Topics

- OCIDateTimeConstruct()
  Constructs a datetime descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.18 OCIDateTimeConstruct()

Constructs a datetime descriptor.

### Purpose

Constructs a datetime descriptor.

### Syntax

```
sword OCIDateTimeConstruct ( void          *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 time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time) 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 a time zone, the date and time fields are assumed to be in the local time of the specified time zone.

If the time zone is not specified, then the session default time zone is assumed.

### Returns

`OCI_SUCCESS`; or `OCI_ERROR`, if datetime is not valid.

### Related Topics

- OCIDateTimeAssign()
  Performs a datetime assignment.
- OCIDateTimeConvert()
  Converts one datetime type to another.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.19 OCIDateTimeConvert()

Converts one datetime type to another.

### Purpose

Converts one datetime type to another.

### Syntax

```
sword OCIDateTimeConvert ( void         *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 a time zone to one with a time zone.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE` if `err` is `NULL`; or `OCI_ERROR`, if the conversion is not possible with the given input values.

### Related Topics

- OCIDateTimeCheck()
  Checks if the given date is valid.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.20 OCIDateTimeFromArray()

Converts an array containing a date to an `OCIDateTime` descriptor.

### Purpose

Converts an array containing a date to an `OCIDateTime` descriptor.

### Syntax

```
sword OCIDateTimeFromArray ( void               *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 is 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`; or `OCI_ERROR` if `type` is invalid.

### Related Topics

- OCIDateTimeFromText()
  Converts the given string to an Oracle datetime type in the `OCIDateTime` descriptor, according to the specified format.
- OCIDateTimeToArray()
  Converts an `OCIDateTime` descriptor to an array.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.21 OCIDateTimeFromText()

Converts the given string to an Oracle datetime type in the `OCIDateTime` descriptor, according to the specified format.

### Purpose

Converts the given string to an Oracle datetime type in the `OCIDateTime` descriptor, according to the specified format.

### Syntax

```
sword OCIDateTimeFromText ( void             *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 occurs 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

See the description of the `TO_DATE` conversion function in the *Oracle Database SQL Language Reference* for a description of the format argument.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE` if `err` is `NULL`; or `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 Topics

- OCIDateTimeToText()
  Converts the given date to a string according to the specified format.
- OCIDateTimeFromArray()
  Converts an array containing a date to an `OCIDateTime` descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.22 OCIDateTimeGetDate()

Gets the date (year, month, day) portion of a datetime value.

### Purpose

Gets the date (year, month, day) portion of a datetime value.

### Syntax

```
sword OCIDateTimeGetDate ( void                *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` or `OCI_ERROR`.

### Related Topics

- OCIDateTimeGetTime()
  Gets the time (hour, min, second, fractional second) of a datetime value.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.23 OCIDateTimeGetTime()

Gets the time (hour, min, second, fractional second) of a datetime value.

### Purpose

Gets the time (hour, min, second, fractional second) of a datetime value.

### Syntax

```
sword OCIDateTimeGetTime ( void          *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 is 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) from a given datetime value.

This function returns an error if the given datetime does not contain time information.

### Returns

`OCI_SUCCESS`; or `OCI_ERROR`, if datetime does not contain time (`SQLT_DATE`).

### Related Topics

- OCIDateTimeGetDate()
  Gets the date (year, month, day) portion of a datetime value.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.24 OCIDateTimeGetTimeZoneName()

Gets the time zone name portion of a datetime value.

### Purpose

Gets the time zone name portion of a datetime value.

### Syntax

```
sword OCIDateTimeGetTimeZoneName ( void              *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) from a given datetime value.

This function returns an error if the given datetime does not contain time information.

### Returns

`OCI_SUCCESS`; or `OCI_ERROR`, if datetime does not contain a time zone (`SQLT_DATE`, `SQLT_TIMESTAMP`).

### Related Topics

- OCIDateTimeGetDate()
  Gets the date (year, month, day) portion of a datetime value.
- OCIDateTimeGetTimeZoneOffset()
  Gets the time zone (hour, minute) portion of a datetime value.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.25 OCIDateTimeGetTimeZoneOffset()

Gets the time zone (hour, minute) portion of a datetime value.

### Purpose

Gets the time zone (hour, minute) portion of a datetime value.

### Syntax

```
sword OCIDateTimeGetTimeZoneOffset ( void              *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 from a given datetime value.

This function returns an error if the given datetime does not contain time information.

### Returns

`OCI_SUCCESS`; or `OCI_ERROR`, if datetime does not contain a time zone (`SQLT_DATE`, `SQLT_TIMESTAMP`).

### Related Topics

- OCIDateTimeGetDate()
  Gets the date (year, month, day) portion of a datetime value.
- OCIDateTimeGetTimeZoneName()
  Gets the time zone name portion of a datetime value.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.26 OCIDateTimeIntervalAdd()

Adds an interval to a datetime to produce a resulting datetime.

### Purpose

Adds an interval to a datetime to produce a resulting datetime.

### Syntax

```
sword OCIDateTimeIntervalAdd ( void         *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 occurs 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 is of same type as the input datetime.

### Returns

`OCI_SUCCESS`, if the function completes successfully; `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer; or `OCI_ERROR`, if the resulting date is before Jan 1, -4713 or is after Dec 31, 9999.

### Related Topics

- OCIDateTimeIntervalSub()
  Subtracts an interval from a datetime and stores the result in a datetime.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.27 OCIDateTimeIntervalSub()

Subtracts an interval from a datetime and stores the result in a datetime.

### Purpose

Subtracts an interval from a datetime and stores the result in a datetime.

### Syntax

```
sword OCIDateTimeIntervalSub ( void         *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 occurs 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 is of same type as the input datetime.

**Returns**

`OCI_SUCCESS`, if the function completes successfully; `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer; or `OCI_ERROR`, if the resulting date is before Jan 1, -4713 or is after Dec 31, 9999.

**Related Topics**

- OCIDateTimeIntervalAdd()
  Adds an interval to a datetime to produce a resulting datetime.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.28 OCIDateTimeSubtract()

Takes two datetimes as input and stores their difference in an interval.

**Purpose**

Takes two datetimes as input and stores their difference in an interval.

**Syntax**

```
sword OCIDateTimeSubtract ( void         *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 (number to be subtracted).

**indate2 (IN)**
Pointer to the minuend (number to be subtracted from).

**inter (OUT)**
Pointer to the output interval.

**Returns**

`OCI_SUCCESS`, if the function completes successfully; `OCI_INVALID_HANDLE` if `err` is `NULL` pointer; or `OCI_ERROR`, if the input datetimes are not of comparable types.

**Related Topics**

- OCIDateTimeCompare()
  Compares two datetime values.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.29 OCIDateTimeSysTimeStamp()

Gets the system current date and time as a time stamp with time zone.

### Purpose

Gets the system current date and time as a time stamp with time zone.

### Syntax

```
sword OCIDateTimeSysTimeStamp ( void           *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 time stamp.

### Returns

`OCI_SUCCESS`; or `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer.

### Related Topics

- OCIDateSysDate()
  Gets the current system date and time of the client.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.30 OCIDateTimeToArray()

Converts an `OCIDateTime` descriptor to an array.

### Purpose

Converts an `OCIDateTime` descriptor to an array.

### Syntax

```
sword OCIDateTimeToArray ( void              *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 the `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`; or `OCI_ERROR`, if datetime is invalid.

**Related Topics**

- OCIDateTimeToText()
  Converts the given date to a string according to the specified format.
- OCIDateTimeFromArray()
  Converts an array containing a date to an `OCIDateTime` descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.31 OCIDateTimeToText()

Converts the given date to a string according to the specified format.

### Purpose

Converts the given date to a string according to the specified format.

### Syntax

```
sword OCIDateTimeToText ( void                *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 occurs 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 value 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

See the description of the `TO_DATE` conversion function in the *Oracle Database SQL Language 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`; or `OCI_ERROR`, if any of the following statements is true:

- The buffer is too small.

- An invalid format is used.
- An unknown language is used.
- There is an overflow error.

**Related Topics**

- OCIDateTimeFromText()
  Converts the given string to an Oracle datetime type in the `OCIDateTime` descriptor, according to the specified format.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.32 OCIDateToText()

Converts a date type to a character string.

### 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; the default language of the 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`.

See the `TO_DATE` conversion function described in the *Oracle Database SQL Language Reference* for a description of format and multilingual arguments.

### Returns

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 Topics

- OCIDateFromText()
  Converts a character string to a date type according to the specified format.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.33 OCIDateZoneToZone()

Converts a date from one time zone to another.

### 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 the 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, see the description of the `NEW_TIME` function in the *Oracle Database SQL Language Reference*. Examples of valid zone strings include:

- AST, Atlantic Standard Time
- ADT, Atlantic Daylight Time
- BST, Bering Standard Time
- BDT, Bering Daylight Time

### Returns

This function returns an error if an invalid date or time zone is passed to it.

### Related Topics

- OCIDateCheck()
  Checks if the given date is valid.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.34 OCIIntervalAdd()

Adds two intervals to produce a resulting interval.

### Purpose

Adds two intervals to produce a resulting interval.

### Syntax

```
sword OCIIntervalAdd ( void         *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_INVALID_HANDLE`, if `err` is a `NULL` pointer; or `OCI_ERROR`, if any of the following statements is true:

- The two input intervals are not mutually comparable.
- The resulting year is greater than `SB4MAXVAL`.
- The resulting year is less than `SB4MINVAL`.

### Related Topics

- OCIIntervalSubtract()
  Subtracts two intervals and stores the result in an interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.35 OCIIntervalAssign()

Copies one interval to another.

### Purpose

Copies one interval to another.

### Syntax

```
void OCIIntervalAssign ( void               *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`; or `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer.

### Related Topics

- OCIIntervalCompare()
  Compares two intervals.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.36 OCIIntervalCheck()

Checks the validity of an interval.

### Purpose

Checks the validity of an interval.

### Syntax

```
sword OCIIntervalCheck ( void                *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)**
Returns zero if the interval is valid; otherwise, it returns the logical operator OR combination of the codes specified in Table 28-9.

**Table 28-9 Error Bits Returned by the valid Parameter for OCIIntervalCheck()**

| 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 |

**Table 28-9 (Cont.) Error Bits Returned by the valid Parameter for OCIIntervalCheck()**

| Macro Name | Bit Number | Error |
|---|---|---|
| `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; or `OCI_ERROR`, on error.

### Related Topics

- OCIIntervalCompare()
  Compares two intervals.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.37 OCIIntervalCompare()

Compares two intervals.

### Purpose

Compares two intervals.

### Syntax

```
sword OCIIntervalCompare (void          *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 as specified in Table 28-10.

**Table 28-10 Comparison Results Returned by the result Parameter for OCIIntervalCompare()**

| 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; or `OCI_ERROR`, if the input values are not mutually comparable.

### Related Topics

- OCIIntervalAssign()
  Copies one interval to another.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.38 OCIIntervalDivide()

Divides an interval by an Oracle `NUMBER` to produce an interval.

### Purpose

Divides an interval by an Oracle `NUMBER` to produce an interval.

### Syntax

```
sword OCIIntervalDivide ( void          *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`; or `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer.

### Related Topics

- OCIIntervalMultiply()
  Multiplies an interval by an Oracle `NUMBER` to produce an interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.39 OCIIntervalFromNumber()

Converts an Oracle `NUMBER` to an interval.

### Purpose

Converts an Oracle `NUMBER` to an interval.

### Syntax

```
sword OCIIntervalFromNumber ( void              *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`; or `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer.

### Related Topics

- OCIIntervalToNumber()
  Converts an interval to an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.40 OCIIntervalFromText()

When given an interval string, returns the interval represented by the string. The type of the interval is the type of the `result` descriptor.

### Purpose

When 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 ( void           *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; or `OCI_ERROR`, if any of the following statements 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).
- The hour is out of range (0 to 11).

- The minutes are out of range (0 to 59).
- The seconds in the minute are out of range (0 to 59).
- The seconds in the day are out of range (0 to 86399).
- The interval is invalid.

**Related Topics**

- OCIIntervalToText()
  When given an interval, produces a string representing the interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.41 OCIIntervalFromTZ()

Returns an `OCI_DTYPE_INTERVAL_DS` of data type `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.

### Purpose

Returns an `OCI_DTYPE_INTERVAL_DS` of data type `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 ( void           *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`; or `OCI_ERROR`, if there is a bad interval type or there are time zone errors.

**Comments**

The input string must be of the form [+/-]TZH:TZM or 'TZR [TZD]'

**Related Topics**

- OCIIntervalFromText()
  When given an interval string, returns the interval represented by the string. The type of the interval is the type of the `result` descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.42 OCIIntervalGetDaySecond()

Gets values of day, hour, minute, and second from an interval.

**Purpose**

Gets values of day, hour, minute, and second from an interval.

**Syntax**

```
sword OCIIntervalGetDaySecond (void               *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 nano seconds.

**interval (IN)**
The input interval.

### Returns

`OCI_SUCCESS`; or `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer.

### Related Topics

- OCIIntervalSetDaySecond()
  Sets day, hour, minute, and second in an interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.43 OCIIntervalGetYearMonth()

Gets year and month from an interval.

### Purpose

Gets year and month from an interval.

### Syntax

```
sword OCIIntervalGetYearMonth ( void                *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`; or `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer.

### Related Topics

- OCIIntervalSetYearMonth()
  Sets year and month in an interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.44 OCIIntervalMultiply()

Multiplies an interval by an Oracle `NUMBER` to produce an interval.

### Purpose

Multiplies an interval by an Oracle `NUMBER` to produce an interval.

### Syntax

```
sword OCIIntervalMultiply ( void                 *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; or `OCI_ERROR`, if any of the following statements is true:

- The resulting year is greater than `SB4MAXVAL`.
- The resulting year is less than `SB4MINVAL`.

**Related Topics**

- OCIIntervalDivide()
  Divides an interval by an Oracle `NUMBER` to produce an interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.45 OCIIntervalSetDaySecond()

Sets day, hour, minute, and second in an interval.

### Purpose

Sets day, hour, minute, and second in an interval.

### Syntax

```
sword OCIIntervalSetDaySecond ( void              *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 nano seconds.

**result (OUT)**
The resulting interval.

### Returns

`OCI_SUCCESS`; or `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer.

### Related Topics

- OCIIntervalGetDaySecond()
  Gets values of day, hour, minute, and second from an interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.46 OCIIntervalSetYearMonth()

Sets year and month in an interval.

### Purpose

Sets year and month in an interval.

### Syntax

```
sword OCIIntervalSetYearMonth ( void               *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; or `OCI_ERROR`, if any of the following statements is true:

- The resulting year is greater than `SB4MAXVAL`.
- The resulting year is less than `SB4MINVAL`.

### Related Topics

- OCIIntervalGetYearMonth()
  Gets year and month from an interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.47 OCIIntervalSubtract()

Subtracts two intervals and stores the result in an interval.

### Purpose

Subtracts two intervals and stores the result in an interval.

### Syntax

```
sword OCIIntervalSubtract ( void         *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; or `OCI_ERROR`, if any of the following statements is true:

- The resulting year is greater than `SB4MAXVAL`.
- The resulting year is less than `SB4MINVAL`.
- The two input intervals are not mutually comparable.

### Related Topics

- OCIIntervalAdd()
  Adds two intervals to produce a resulting interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.48 OCIIntervalToNumber()

Converts an interval to an Oracle `NUMBER`.

### Purpose

Converts an interval to an Oracle `NUMBER`.

### Syntax

```
sword OCIIntervalToNumber ( void               *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`; or `OCI_INVALID_HANDLE`, if `err` is a `NULL` pointer.

**Related Topics**

- OCIIntervalFromNumber()
  Converts an Oracle `NUMBER` to an interval.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.3.49 OCIIntervalToText()

When given an interval, produces a string representing the interval.

### Purpose

When given an interval, produces a string representing the interval.

### Syntax

```
sword OCIIntervalToText ( void                *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; or `OCI_ERROR`, if the buffer is not large enough to hold the result.

**Related Topics**

- OCIIntervalFromText()
  When given an interval string, returns the interval represented by the string. The type of the interval is the type of the `result` descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4 OCI NUMBER Functions

Lists and describes the OCI `NUMBER` functions.

Table 28-11 describes the OCI `NUMBER` functions that are described in this section.

> **See Also:**
>
> OCINumber Examples

**Table 28-11 NUMBER Functions**

| Function | Purpose |
|---|---|
| OCINumberAbs() | Compute the absolute value |
| OCINumberAdd() | Add a `NUMBER` to a `NUMBER` |
| OCINumberArcCos() | Compute the arc cosine |

**Table 28-11 (Cont.) NUMBER Functions**

| Function | Purpose |
|---|---|
| OCINumberArcSin() | Compute the arc sine |
| OCINumberArcTan() | Compute the arc tangent |
| OCINumberArcTan2() | Compute the arc tangent given two arguments for `NUMBER` |
| OCINumberAssign() | Assign one `NUMBER` to another |
| OCINumberCeil() | Compute the ceiling of `NUMBER` |
| OCINumberCmp() | Compare one `NUMBER` with another `NUMBER` |
| OCINumberCos() | Compute the cosine |
| OCINumberDec() | Decrement a `NUMBER` |
| OCINumberDiv() | Divide a `NUMBER` by a `NUMBER` |
| OCINumberExp() | Raise *e* to the specified Oracle `NUMBER` power |
| OCINumberFloor() | Compute the floor value of a `NUMBER` |
| OCINumberFromInt() | Convert an integer to an Oracle `NUMBER` |
| OCINumberFromReal() | Convert a real type to an Oracle `NUMBER` |
| OCINumberFromText() | Convert a string to an Oracle `NUMBER` |
| OCINumberHypCos() | Compute the hyperbolic cosine |
| OCINumberHypSin() | Compute the hyperbolic sine |
| OCINumberHypTan() | Compute the hyperbolic tangent |
| OCINumberInc() | Increment an Oracle `NUMBER` |
| OCINumberIntPower() | Raise a given base to an integer power |
| OCINumberIsInt() | Test if a `NUMBER` is an integer |
| OCINumberIsZero() | Test if a `NUMBER` is zero |
| OCINumberLn() | Compute the natural logarithm |
| OCINumberLog() | Compute the logarithm to an arbitrary base |
| OCINumberMod() | Gets the modulus (remainder) of the division of two Oracle `NUMBER`s |
| OCINumberMul() | Multiply one Oracle `NUMBER` by another Oracle `NUMBER` |
| OCINumberNeg() | Negates an Oracle `NUMBER` |
| OCINumberPower() | Raises a given base to a given exponent |
| OCINumberPrec() | Round a `NUMBER` to a specified number of decimal places |
| OCINumberRound() | Round an Oracle `NUMBER` to a specified decimal place |
| OCINumberSetPi() | Initialize a `NUMBER` to pi |
| OCINumberSetZero() | Initialize a `NUMBER` to zero |

**Table 28-11 (Cont.) NUMBER Functions**

| Function | Purpose |
|---|---|
| OCINumberShift() | Multiply by 10, shifting a specified number of decimal places |
| OCINumberSign() | Obtain the sign of an Oracle `NUMBER` |
| OCINumberSin() | Compute the sine |
| OCINumberSqrt() | Compute the square root of a `NUMBER` |
| OCINumberSub() | Subtract a `NUMBER` from a `NUMBER` |
| OCINumberTan() | Compute the tangent |
| OCINumberToInt() | Convert an Oracle `NUMBER` to an integer |
| OCINumberToReal() | Convert an Oracle `NUMBER` to a real type |
| OCINumberToRealArray() | Convert an array of `NUMBER` to a real array. |
| OCINumberToText() | Convert an Oracle `NUMBER` to a string |
| OCINumberTrunc() | Truncate an Oracle `NUMBER` at a specified decimal place |

## 28.4.1 OCINumberAbs()

Computes the absolute value of an Oracle `NUMBER`.

### 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`.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.2 OCINumberAdd()

Adds a `NUMBER` to another `NUMBER`.

### Purpose

Adds a `NUMBER` to another `NUMBER`.

### 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)**
The first `NUMBER` to be added to the second `NUMBER`.

**result (OUT)**
Result of adding `number1` to `number2`.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberSub()
  Subtracts one Oracle `NUMBER` from another Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.3 OCINumberArcCos()

Takes the arc cosine in radians of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`, if `number` is less than –1, or if `number` is greater than 1.

**Related Topics**

- OCINumberCos()
  Computes the cosine in radians of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.4 OCINumberArcSin()

Takes the arc sine in radians of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`, if `number` is less than –1, or if `number` is greater than 1.

**Related Topics**

- OCINumberSin()
  Computes the sine in radians of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.5 OCINumberArcTan()

Takes the arc tangent in radians of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

**Related Topics**

- OCINumberTan()
  Computes the tangent in radians of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.6 OCINumberArcTan2()

Takes the arc tangent given two arguments for Oracle `NUMBER`.

### Purpose

Takes the arc tangent given two arguments for Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL` or if `number2` equals 0.

### Related Topics

- OCINumberTan()
  Computes the tangent in radians of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.7 OCINumberAssign()

Assigns one Oracle `NUMBER` to another Oracle `NUMBER`.

### 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`.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberCmp()
  Compares one Oracle `NUMBER` with another Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.8 OCINumberCeil()

Computes the ceiling value of an Oracle `NUMBER`.

### 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 that contains the ceiling value of the input `NUMBER`.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberFloor()
  Computes the floor (round down) value of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.9 OCINumberCmp()

Compares one Oracle `NUMBER` with another Oracle `NUMBER`.

### Purpose

Compares one Oracle `NUMBER` with another Oracle `NUMBER`.

### 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)**
The first `NUMBER` to compare with the second `NUMBER`.

**result (OUT)**
Comparison result as specified in Table 28-12.

**Table 28-12 Comparison Results Returned by the result Parameter for OCINumberCmp()**

| Comparison Result | Output in result Parameter |
|---|---|
| `number1` < `number2` | negative |
| `number1` = `number2` | `0` |
| `number1` > `number2` | positive |

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberAssign()
  Assigns one Oracle `NUMBER` to another Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.10 OCINumberCos()

Computes the cosine in radians of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberArcCos()
  Takes the arc cosine in radians of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.11 OCINumberDec()

12.1.0.2, added a short description.

### Purpose

Decrements an Oracle `NUMBER` in place.

### 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 input is too large, it is treated as 0; the result is an Oracle `NUMBER` 1. If the input is not a positive integer, the result is unpredictable.

### Returns

This function returns an error if the input `NUMBER` is `NULL`.

### Related Topics

- OCINumberInc()
  Increments an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.12 OCINumberDiv()

Divides one Oracle `NUMBER` by another Oracle `NUMBER`.

### Purpose

Divides one Oracle `NUMBER` by another Oracle `NUMBER`.

### 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 the result in `result`.

### Returns

This function returns an error if any of the following statements is true:

- Any of the `NUMBER` arguments is `NULL`.
- There is an underflow error.
- There is a divide-by-zero error.

### Related Topics

- OCINumberMul()
  Multiplies one Oracle `NUMBER` by another Oracle `NUMBER`.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.13 OCINumberExp()

Raises *e* to the specified Oracle `NUMBER` power.

### 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.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberLn()
  Takes the natural logarithm (base *e*) of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.14 OCINumberFloor()

Computes the floor (round down) value of an Oracle `NUMBER`.

### Purpose

Computes the floor (round down) 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 (round down) value of the input `NUMBER`.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberCeil()
  Computes the ceiling value of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.15 OCINumberFromInt()

Converts an integer to an Oracle `NUMBER`.

### Purpose

Converts an integer to an Oracle `NUMBER`.

### Syntax

```
sword OCINumberFromInt ( OCIError           *err,
                         const void         *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 system-native integer type, such as `ub4` or `sb2`, to an Oracle `NUMBER`.

### Returns

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 Topics

- OCINumberToInt()
  Converts an Oracle `NUMBER` type to integer.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.16 OCINumberFromReal()

Converts a real (floating-point) type to an Oracle `NUMBER`.

### Purpose

Converts a real (floating-point) type to an Oracle `NUMBER`.

### Syntax

```
sword OCINumberFromReal ( OCIError           *err,
                          const void         *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 system-native floating-point type to an Oracle `NUMBER`.

### Returns

This function returns an error if `number` or `rnum` is `NULL`, or if `rnum_length` equals zero.

**Related Topics**

- OCINumberFromText()
  Converts a character string to an Oracle `NUMBER`.

## 28.4.17 OCINumberFromText()

Converts a character string to an Oracle `NUMBER`.

### Purpose

Converts a character string to an 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)**
Globalization support format specification. If it is the `NULL` string (""), then the default parameters for the session are 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. See the `TO_NUMBER` conversion function described in the *Oracle Database SQL Language Reference* for a description of format and multilingual parameters.

### Returns

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.

## 28.4.18 OCINumberHypCos()

Computes the hyperbolic cosine of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if either of the `number` arguments is `NULL`.

> **Note:**
>
> An Oracle `NUMBER` overflow causes an unpredictable result value.

### Related Topics

- OCINumberHypSin()
  Computes the hyperbolic sine of an Oracle `NUMBER`.
- OCINumberHypTan()
  Computes the hyperbolic tangent of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.19 OCINumberHypSin()

Computes the hyperbolic sine of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if either of the `NUMBER` arguments is `NULL`.

> **Note:**
>
> An Oracle `NUMBER` overflow causes an unpredictable result value.

### Related Topics

- OCINumberHypCos()
  Computes the hyperbolic cosine of an Oracle `NUMBER`.
- OCINumberHypTan()
  Computes the hyperbolic tangent of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.20 OCINumberHypTan()

Computes the hyperbolic tangent of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if either of the `NUMBER` arguments is `NULL`.

> **Note:**
>
> An Oracle `NUMBER` overflow causes an unpredictable result value.

### Related Topics

- OCINumberHypCos()
  Computes the hyperbolic cosine of an Oracle `NUMBER`.
- OCINumberHypSin()
  Computes the hyperbolic sine of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.21 OCINumberInc()

Increments an Oracle `NUMBER`.

### Purpose

Increments an Oracle `NUMBER`.

### 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 input is too large, it is treated as 0 - the result is an Oracle `NUMBER` 1. If the input is not a positive integer, the result is unpredictable.

### Returns

This function returns an error if the input `NUMBER` is `NULL`.

### Related Topics

- OCINumberDec()
  12.1.0.2, added a short description.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.22 OCINumberIntPower()

Raises a given base to a given integer power.

### 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.

### Returns

This function returns an error if either of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberPower()
  Raises a given base to a given exponent.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.23 OCINumberIsInt()

Tests if an OCINumber is an integer.

### 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; otherwise, `FALSE`

### Returns

This function returns an error if `number` or `result` is `NULL`.

### Related Topics

- OCINumberRound()
  Rounds an Oracle `NUMBER` to a specified decimal place.
- OCINumberTrunc()
  Truncates an Oracle `NUMBER` at a specified decimal place.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.24 OCINumberIsZero()

Tests if the given `NUMBER` equals zero.

### Purpose

Tests if the given `NUMBER` equals 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`.

### Returns

This function returns an error if the `NUMBER` argument is `NULL`.

### Related Topics

- OCINumberSetZero()
  Initializes an Oracle `NUMBER` to zero.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.25 OCINumberLn()

Takes the natural logarithm (base *e*) of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if either of the `NUMBER` arguments is `NULL`, or if `number` is less than or equal to zero.

### Related Topics

- OCINumberExp()
  Raises *e* to the specified Oracle `NUMBER` power.
- OCINumberLog()
  Takes the logarithm, to any base, of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.26 OCINumberLog()

Takes the logarithm, to any base, of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if:

- Any of the `NUMBER` arguments is `NULL`
- The value of `number` <= 0
- The value of `base` <= 0

### Related Topics

- OCINumberLn()
  Takes the natural logarithm (base *e*) of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.27 OCINumberMod()

Gets the modulus (remainder) of the division of two Oracle `NUMBER`s.

### Purpose

Gets the modulus (remainder) of the division of two Oracle `NUMBER`s.

### 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.

### Returns

This function returns an error if `number1` or `number2` is `NULL`, or if there is a divide-by-zero error.

### Related Topics

- OCINumberDiv()
  Divides one Oracle `NUMBER` by another Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.28 OCINumberMul()

Multiplies one Oracle `NUMBER` by another Oracle `NUMBER`.

### Purpose

Multiplies one Oracle `NUMBER` by another Oracle `NUMBER`.

### 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 the result in `result`.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

**Related Topics**

- OCINumberDiv()
  Divides one Oracle `NUMBER` by another Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.29 OCINumberNeg()

Negates an Oracle `NUMBER`.

### 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`.

### Returns

This function returns an error if either of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberAbs()
  Computes the absolute value of an Oracle `NUMBER`.
- OCINumberSign()
  Gets sign of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.30 OCINumberPower()

Raises a given base to a given exponent.

### 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.

**Returns**

This function returns an error if any of the `NUMBER` arguments is `NULL`.

**Related Topics**

- OCINumberExp()
  Raises *e* to the specified Oracle `NUMBER` power.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.31 OCINumberPrec()

Rounds an `OCINumber` to a specified number of decimal digits.

**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.

**Returns**

This function returns an error any of the `NUMBER` arguments is `NULL`.

**Related Topics**

- OCINumberRound()
  Rounds an Oracle `NUMBER` to a specified decimal place.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.32 OCINumberRound()

Rounds an Oracle `NUMBER` to a specified decimal place.

### 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.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberTrunc()
  Truncates an Oracle `NUMBER` at a specified decimal place.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.33 OCINumberSetPi()

Sets an `OCINumber` to pi.

### 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 Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.34 OCINumberSetZero()

Initializes an Oracle `NUMBER` to zero.

### 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 never produces an error.

**num (IN/OUT)**
Oracle `NUMBER` to initialize to zero value.

### Comments

None.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.35 OCINumberShift()

Multiplies a `NUMBER` by a power of 10 by shifting it a specified number of decimal places.

### 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.

### Returns

This function returns an error if the input `number` is `NULL`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.36 OCINumberSign()

Gets sign of an Oracle `NUMBER`.

### 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)**
Oracle `NUMBER` whose sign is returned.

**result (OUT)**
Table 28-13 lists the possible return values.

**Table 28-13 Values of result**

| Value of *number* | Output in result Parameter |
|---|---|
| `number` < 0 | -1 |
| `number` == 0 | 0 |
| `number` > 0 | 1 |

### Returns

This function returns an error if `number` or `result` is `NULL`.

### Related Topics

- OCINumberAbs()
  Computes the absolute value of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.37 OCINumberSin()

Computes the sine in radians of an Oracle `NUMBER`.

### 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.

#### Returns

This function returns an error if either of the `number` arguments is `NULL`.

#### Related Topics

- OCINumberArcSin()
  Takes the arc sine in radians of an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.38 OCINumberSqrt()

Computes the square root of an Oracle `NUMBER`.

#### 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 that contains the square root of the input `NUMBER`.

#### Returns

This function returns an error if `number` is `NULL` or `number` is negative.

#### Related Topics

- OCINumberPower()
  Raises a given base to a given exponent.

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.39 OCINumberSub()

Subtracts one Oracle `NUMBER` from another Oracle `NUMBER`.

### Purpose

Subtracts one Oracle `NUMBER` from another Oracle `NUMBER`.

### 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 the result in `result`.

### Returns

This function returns an error if any of the `number` arguments is `NULL`.

**Related Topics**

- OCINumberAdd()
  Adds a `NUMBER` to another `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.40 OCINumberTan()

Computes the tangent in radians of an Oracle `NUMBER`.

### 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.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberArcTan()
  Takes the arc tangent in radians of an Oracle `NUMBER`.
- OCINumberArcTan2()
  Takes the arc tangent given two arguments for Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.41 OCINumberToInt()

Converts an Oracle `NUMBER` type to integer.

### Purpose

Converts an Oracle `NUMBER` type to integer.

### Syntax

```
sword OCINumberToInt ( OCIError              *err,
                       const OCINumber       *number,
                       uword                 rsl_length,
                       uword                 rsl_flag,
                       void                  *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)**
Oracle `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`.

### Returns

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 Topics

- OCINumberFromInt()
  Converts an integer to an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.42 OCINumberToReal()

Converts an Oracle `NUMBER` type to a real type.

### Purpose

Converts an Oracle `NUMBER` type to a real type.

### Syntax

```
sword OCINumberToReal ( OCIError           *err,
                        const OCINumber    *number,
                        uword               rsl_length,
                        void               *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)**
Oracle `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 system-native real type. This function only converts `NUMBER`s up to LDBL_DIG, DBL_DIG, or

FLT_DIG digits of precision and removes trailing zeros. These constants are defined in `float.h`.

You must pass a valid `OCINumber` to this function. Otherwise, the result is undefined.

**Related Topics**

- OCINumberFromReal()
  Converts a real (floating-point) type to an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.43 OCINumberToRealArray()

Converts an array of `NUMBER` to an array of real type.

### Purpose

Converts an array of `NUMBER` to an array of real type.

### Syntax

```
sword OCINumberToRealArray ( OCIError          *err,
                             const OCINumber   **number,
                             uword             elems,
                             uword             rsl_length,
                             void              *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 system-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`, if the function completes successfully; `OCI_INVALID_HANDLE`, if `err` is `NULL`; or `OCI_ERROR`, if `number` or `rsl` is `NULL` or `rsl_length` is 0.

**Related Topics**

- OCINumberToReal()
  Converts an Oracle `NUMBER` type to a real type.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.44 OCINumberToText()

Converts an Oracle `NUMBER` to a character string according to a specified format.

**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,
                        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()`.

**number (IN)**
Oracle `NUMBER` to convert.

**fmt (IN)**
Conversion format.

**fmt_length (IN)**
Length of the `fmt` parameter.

**nls_params (IN)**
Globalization 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

See the `TO_NUMBER` conversion function described in the *Oracle Database SQL Language Reference*for a description of format and globalization support parameters.

The converted number string is stored in `buf`, up to a maximum of `buf_size` bytes.

### Returns

This function returns an error if:

- The value of `number` or `buf` is `NULL`
- The buffer is too small
- An invalid format or invalid multibyte format is passed
- A number to text translation for given format causes an overflow

**Related Topics**

- OCINumberFromText()
  Converts a character string to an Oracle `NUMBER`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.4.45 OCINumberTrunc()

Truncates an Oracle `NUMBER` at a specified decimal place.

### 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.

### Returns

This function returns an error if any of the `NUMBER` arguments is `NULL`.

### Related Topics

- OCINumberRound()
  Rounds an Oracle `NUMBER` to a specified decimal place.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 28.5 OCI Raw Functions

Lists and describes the OCI raw functions.

Table 28-14 describes the OCI raw functions that are described in this section.

**Table 28-14 Raw Functions**

| Function | Purpose |
|---|---|
| OCIRawAllocSize() | Get allocated size of raw memory in bytes |
| OCIRawAssignBytes() | Assign raw bytes to raw |
| OCIRawAssignRaw() | Assign raw to raw |
| OCIRawPtr() | Get raw data pointer |
| OCIRawResize() | Resize memory of variable-length raw |
| OCIRawSize() | Get raw size |

## 28.5.1 OCIRawAllocSize()

Gets the allocated size of raw memory in bytes.

### Purpose

Gets the 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**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 that is returned.

**Comments**

The allocated size is greater than or equal to the actual raw size.

**Related Topics**

- OCIRawResize()
  Resizes the memory of a given variable-length `raw`.
- OCIRawSize()
  Returns the size of a given raw in bytes.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.5.2 OCIRawAssignBytes()

Assigns raw bytes of type `ub1*` to Oracle `OCIRaw*` data type.

**Purpose**

Assigns raw bytes of type `ub1*` to Oracle `OCIRaw*` data type.

**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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**rhs (IN)**
Right-hand side (source) of the assignment, of data type `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 data type. The `lhs` raw may be resized depending upon the size of the `rhs`. The raw bytes assigned are of type `ub1`.

**Related Topics**

- OCIRawAssignRaw()
  Assigns one Oracle `RAW` data type to another Oracle `RAW` data type.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.5.3 OCIRawAssignRaw()

Assigns one Oracle `RAW` data type to another Oracle `RAW` data type.

**Purpose**

Assigns one Oracle `RAW` data type to another Oracle `RAW` data type.

**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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**rhs (IN)**
Right-hand side (`rhs`) (source) of the assignment; `OCIRaw` data.

**lhs (IN/OUT)**
Left-hand side (`lhs`) (target) of the assignment; `OCIRaw` data.

#### Comments

Assigns `rhs` `OCIRaw` to `lhs` `OCIRaw`. The `lhs` `OCIRaw` may be resized depending upon the size of the `rhs`.

#### Related Topics

- OCIRawAssignBytes()
  Assigns raw bytes of type `ub1*` to Oracle `OCIRaw*` data type.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.5.4 OCIRawPtr()

Gets the pointer to raw data.

#### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**raw (IN)**
Pointer to the data of a given `raw`.

#### Comments

None.

**Related Topics**

- OCIRawAssignBytes()
  Assigns raw bytes of type `ub1*` to Oracle `OCIRaw*` data type.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.5.5 OCIRawResize()

Resizes the memory of a given variable-length `raw`.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**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 is freed. If the input raw is `NULL` (`raw` == `NULL`), then this function allocates 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 Topics**

- OCIRawAllocSize()
  Gets the allocated size of raw memory in bytes.
- OCIRawSize()
  Returns the size of a given raw in bytes.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.5.6 OCIRawSize()

Returns the size of a given raw in bytes.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**raw (IN/OUT)**
Raw whose size is returned.

### Comments

None.

**Related Topics**

- OCIRawAllocSize()
  Gets the allocated size of raw memory in bytes.
- OCIRawResize()
  Resizes the memory of a given variable-length `raw`.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.6 OCI REF Functions

Lists and describes the OCI Reference (`REF`) functions.

Table 28-15 describes the OCI Reference (`REF`) functions that are described in this section.

**Table 28-15 Ref Functions**

| Function | Purpose |
|---|---|
| OCIRefAssign() | Assign one REF to another |
| OCIRefClear() | Clear or nullify a REF |
| OCIRefFromHex() | Convert hexadecimal string to REF |
| OCIRefHexSize() | Return size of hexadecimal representation of REF |
| OCIRefIsEqual() | Compare two REFs for equality |
| OCIRefIsNull() | Test if a REF is `NULL` |
| OCIRefToHex() | Convert REF to hexadecimal string |

### 28.6.1 OCIRefAssign()

Assigns one `REF` to another, such that both reference the same object.

**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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**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()` allocates memory for the `target REF` in the OCI object cache before the copy operation.

### Related Topics

- OCIRefIsEqual()
  Compares two `REF`s to determine if they are equal.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.6.2 OCIRefClear()

Clears or `NULL`ifies a given `REF`.

### Purpose

Clears or `NULL`ifies a given `REF`.

### Syntax

```
void OCIRefClear ( OCIEnv        *env,
                   OCIRef        *ref );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**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 nonoperational.

### Related Topics

- OCIRefIsNull()
  Tests if a `REF` is `NULL`.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.6.3 OCIRefFromHex()

Converts the given hexadecimal string into a `REF`.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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, 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 reused.

### 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.

**Related Topics**

- OCIRefToHex()
  Converts a `REF` to a hexadecimal string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.6.4 OCIRefHexSize()

Returns the size of the hexadecimal representation of a `REF`.

### Purpose

Returns the size of the hexadecimal 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**ref (IN)**
`REF` whose size in hexadecimal representation in bytes is returned.

### 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.

### Returns

The size of the hexadecimal representation of the `REF`.

### Related Topics

- OCIRefFromHex()
  Converts the given hexadecimal string into a `REF`.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.6.5 OCIRefIsEqual()

Compares two `REF`s to determine if they are equal.

### Purpose

Compares two `REF`s 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**x (IN)**
`REF` to compare.

**y (IN)**
`REF` to compare.

**Comments**

Two `REF`s are equal if and only if they are both referencing the same object, whether persistent or transient.

> **Note:**
>
> Two `NULL` `REF`s are considered *not* equal by this function.

**Returns**

`TRUE`, if the two `REF`s are equal.

`FALSE`, if the two `REF`s are not equal, or `x` is `NULL`, or `y` is `NULL`.

**Related Topics**

- OCIRefAssign()
  Assigns one `REF` to another, such that both reference the same object.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.6.6 OCIRefIsNull()

Tests if a `REF` is `NULL`.

**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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**ref (IN)**
`REF` to test for `NULL`.

**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.

**Returns**

Returns `TRUE` if the given `REF` is `NULL`; otherwise, it returns `FALSE`.

**Related Topics**

- OCIRefClear()
  Clears or `NULL`ifies a given `REF`.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.6.7 OCIRefToHex()

Converts a `REF` to a hexadecimal string.

**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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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)**
`REF` to be converted into a hexadecimal string; if `ref` is a `NULL REF` (that is, `OCIRefIsNull(ref)` == `TRUE`), then a zero `hex_length` value is returned.

**hex (OUT)**
Buffer that is large enough to contain the resulting hexadecimal string; the content 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.

### Returns

This function returns an error if the given buffer is not big enough to hold the resulting string.

**Related Topics**

- OCIRefFromHex()
  Converts the given hexadecimal string into a `REF`.
- OCIRefHexSize()
  Returns the size of the hexadecimal representation of a `REF`.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 28.7 OCI String Functions

Lists and describes the OCI string functions.

Table 28-16 describes the OCI string functions that are described in this section.

**Table 28-16    String Functions**

| Function | Purpose |
|---|---|
| OCIStringAllocSize() | Get the allocated size of string memory in bytes |
| OCIStringAssign() | Assign a string to a string |
| OCIStringAssignText() | Assign a text string to a string |
| OCIStringPtr() | Get a string pointer |
| OCIStringResize() | Resize the string memory |
| OCIStringSize() | Get the string size |

## 28.7.1 OCIStringAllocSize()

Gets the allocated size of string memory in code points (Unicode) or in bytes.

### Purpose

Gets the allocated size of string memory in code points (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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**vs (IN)**
String whose allocated size in bytes is returned. The `vs` parameter 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 Topics**

- OCIStringResize()
  Resizes the memory of a given string.
- OCIStringSize()
  Gets the size of the given string `vs`.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.7.2 OCIStringAssign()

Assigns one string to another string.

**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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**rhs (IN)**
Right-hand side (source) of the assignment. Can be in UTF-16 format.

**lhs (IN/OUT)**
Left-hand side (target) of the assignment. Its buffer is in UTF-16 format if `rhs` is UTF-16.

### Comments

Assigns `rhs` string to `lhs` string. The `lhs` string can be resized depending upon the size of the `rhs`. The assigned string is `NULL`-terminated. The length field does not include the extra code point or byte needed for `NULL`-termination.

### Returns

This function returns an error if the assignment operation runs out of space.

### Related Topics

- OCIStringAssignText()
  Assigns the source text string to the target string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.7.3 OCIStringAssignText()

Assigns the source text string to the target string.

### Purpose

Assigns the source text string to the target string.

### Syntax

```
sword OCIStringAssignText ( OCIEnv           *env,
                            OCIError         *err,
                            const OraText    *rhs,
                            ub4               rhs_len,
                            OCIString       **lhs );
```

### Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode. See the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**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 in Unicode if `rhs` is in 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 does not include the extra byte or code point needed for `NULL`-termination.

### Related Topics

- OCIStringAssign()
  Assigns one string to another string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.7.4 OCIStringPtr()

Gets a pointer to the text of a given string.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**vs (IN)**
Pointer to the `OCIString` object whose character string is returned. If `vs` is in UTF-16 format, the returned buffer is also in UTF-16 format. To determine the encoding of the returned buffer, check the UTF-16 information in the `OCIString` `vs` itself, because it is not guaranteed that a particular `OCIString` will have the same setting as `env` does. Check an object OCI function that is designed to check member fields in objects.

### Comments

None.

### Related Topics

- OCIStringAssign()
  Assigns one string to another string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.7.5 OCIStringResize()

Resizes the memory of a given string.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**new_size (IN)**
New memory size of the string in bytes. The `new_size` parameter must include space for the `NULL` character as the string terminator.

**str (IN/OUT)**
Allocated memory for the string that 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 Topics**

- OCIStringAllocSize()
  Gets the allocated size of string memory in code points (Unicode) or in bytes.
- OCIStringSize()
  Gets the size of the given string `vs`.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.7.6 OCIStringSize()

Gets the size of the given string `vs`.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) for more information.

**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 Topics**

- OCIStringResize()
  Resizes the memory of a given string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 28.8 OCI Table Functions

Lists and describes the OCI table functions.

Table 28-17 describes the OCI table functions that are described in this section.

**Table 28-17    Table Functions**

| Function | Purpose |
|---|---|
| OCITableDelete() | Delete element |
| OCITableExists() | Test whether element exists |
| OCITableFirst() | Return first index of table |
| OCITableLast() | Return last index of table |
| OCITableNext() | Return next available index of table |
| OCITablePrev() | Return previous available index of table |
| OCITableSize() | Return current size of table |

## 28.8.1 OCITableDelete()

Deletes the element at the specified index.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**index (IN)**
Index of the element that 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.

> **Note:**
>
> The position ordinals of the remaining elements of the table are not changed by `OCITableDelete()`. The delete operation creates *holes* in the table.

### Returns

An error is also returned if any input parameter is `NULL`.

### Related Topics

- OCITableExists()
  Tests whether an element exists at the given index.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.8.2 OCITableExists()

Tests whether an element exists at the given index.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**tbl (IN)**
Table in which the given index is checked.

**index (IN)**
Index of the element that is checked for existence.

**exists (OUT)**
Set to `TRUE` if the element at the given `index` exists; otherwise, it is set to `FALSE`.

### Returns

This function returns an error if any input parameter is `NULL`.

### Related Topics

- OCITableDelete()
  Deletes the element at the specified index.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.8.3 OCITableFirst()

Returns the index of the first existing element in a given table.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**tbl (IN)**
Table to scan.

**index (OUT)**
First index of the element that exists in the given table that is returned.

### Comments

If `OCITableDelete()` deletes the first five elements of a table, `OCITableFirst()` returns the value 6.

> **See Also:**
>
> OCITableDelete() for information regarding non-data *holes* (deleted elements) in tables

### Returns

This function returns an error if the table is empty.

### Related Topics

- OCITableDelete()
  Deletes the element at the specified index.
- OCITableLast()
  Returns the index of the last existing element of a table.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.8.4 OCITableLast()

Returns the index of the last existing element of a table.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**tbl (IN)**
Table to scan.

**index (OUT)**
Index of the last existing element in the table.

### Comments

`OCITableLast()` returns the largest index numbered element in the index by integer collection.

### Returns

This function returns an error if the table is empty.

### Related Topics

- OCITableFirst()
  Returns the index of the first existing element in a given table.
- OCITableNext()
  Returns the index of the next existing element of a table.
- OCITablePrev()
  Returns the index of the previous existing element of a table.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 28.8.5 OCITableNext()

Returns the index of the next existing element of a table.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**index (IN)**
Index for the starting point of the 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; otherwise, `TRUE`.

### Returns

Returns the smallest position `j`, greater than `index`, such that `exists(j)` is `TRUE`.

### Related Topics

- OCITablePrev()
  Returns the index of the previous existing element of a table.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> The description of OCITableSize() for information about the existence of non-data *holes* (deleted elements) in tables

## 28.8.6 OCITablePrev()

Returns the index of the previous existing element of a table.

### 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**index (IN)**
Index for the starting point of the 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; otherwise, `TRUE`.

### Returns

Returns the largest position `j`, less than `index`, such that `exists (j)` is `TRUE`.

### Related Topics

- OCITableNext()
  Returns the index of the next existing element of a table.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

> **See Also:**
>
> The description of OCITableSize() for information about the existence of non-data *holes* (deleted elements) in tables

## 28.8.7 OCITableSize()

Returns the size of the given table, not including any holes created by deleted elements.

### Purpose

Returns the size of the given table, not including any holes created by 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 the descriptions of `OCIEnvCreate()`, `OCIEnvNlsCreate()`, and `OCIInitialize()` (deprecated) 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()`.

**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 is decremented when elements are deleted from the nested table. So this count does not include any *holes* created by deleting elements. To get the count including the holes created by the deleted elements, use `OCICollSize()`.

The following code example shows a code fragment where an element is deleted from a nested table.

Deleting an Element from a Nested table

```
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()`, as shown in the following code example. Continuing the previous code example.

Getting a Count of All Elements Including Deleted Elements from a Nested Table

```
OCICollSize(...)
// 'size' returned is still equal to 5
```

### Returns

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 Topics

- OCICollSize()
  Gets the current size in number of elements of the given collection.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIInitialize()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 29

# OCI Cartridge Functions

This chapter presents the cartridge functions.

For code examples, see the demonstration programs included with your Oracle Database installation. For additional information, see OCI Demonstration Programs.

This chapter contains these topics:

- Introduction to External Procedure and Cartridge Services Functions
- Cartridge Services — OCI External Procedures
- Cartridge Services — Memory Services
- Cartridge Services — Maintaining Context
- Cartridge Services — Parameter Manager Interface
- Cartridge Services — File I/O Interface
- Cartridge Services — String Formatting Interface

## 29.1 Introduction to External Procedure and Cartridge Services Functions

This section introduces the OCI external procedure functions.

These functions enable users of external procedures to raise errors, allocate some memory, and get OCI context information.

Then the cartridge services functions are described.

**See Also:**

- *Oracle Database Development Guide* for more information about using these functions in external procedures
- *Oracle Database Data Cartridge Developer's Guide* for more information about using these functions

### 29.1.1 Conventions for OCI Functions

Describes conventions used in describing each function.

See the Conventions for OCI Functions for the conventions used in describing each function. The entries for each function may also contain the following information:

- Return Codes
- With_Context Type

### 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 follows:

```
OCIExtProcContext *with_context;
```

# 29.2 Cartridge Services — OCI External Procedures

Lists and describes cartridge services OCI external procedure functions.

Table 29-1 lists the OCI external procedure functions for C that are described in this section.

**Table 29-1 External Procedures Functions**

| Function | Purpose |
|---|---|
| OCIExtProcAllocCallMemory() | Allocate memory for the duration of the External Procedure |
| OCIExtProcGetEnv() | Get the OCI environment, service context, and error handles |
| OCIExtProcRaiseExcp() | Raise an Exception to PL/SQL |
| OCIExtProcRaiseExcpWithMsg() | Raise an exception with a message |

## 29.2.1 OCIExtProcAllocCallMemory()

Allocates *N* bytes of memory for the duration of the external procedure.

### Purpose

Allocates *N* bytes of memory for the duration of the external procedure.

### Syntax

```
void  * OCIExtProcAllocCallMemory ( OCIExtProcContext    *with_context,
                                    size_t                amount );
```

### Parameters

**with_context (IN)**
The `with_context` pointer that is passed to the C external procedure.
`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 follows:

```
OCIExtProcContext *with_context;
```

**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

Using OCIExtProcAllocCallMemory() to Allocate 1024 Bytes of Memory

```
text *ptr = (text *)OCIExtProcAllocCallMemory(wctx, 1024)
```

**Related Topics**

- OCIMemoryAlloc()
  Allocates memory of a given size from a given duration.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.2.2 OCIExtProcGetEnv()

Gets the OCI environment, service context, and error handles.

### 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.
`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 follows:

```
OCIExtProcContext *with_context;
```

**envh (OUT)**
Pointer to a variable to store the OCI environment handle.

**svch (OUT)**
Pointer to a variable to store the OCI service handle.

**errh (OUT)**
Pointer to a variable to store 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.

Example of a call:

```
OCIEnv    *envh;
OCISvcCtx *svch;
OCIError  *errh;
...
OCIExtProcGetEnv(ctx,&envh,&svch,&errh);
```

### Returns

This function returns `OCI_SUCCESS` if the call succeeds; otherwise, it returns `OCI_ERROR`.

### Related Topics

- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCIArrayDescriptorAlloc()
  Allocates an array of descriptors.
- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.

## 29.2.3 OCIExtProcRaiseExcp()

Raises an Exception to PL/SQL.

### Purpose

Raises 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.
`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 follows:

```
OCIExtProcContext *with_context;
```

**errnum (IN)**
Oracle Database error number to signal to PL/SQL. The `errnum` value must be a positive number and in the range 1 to 32767.

### Comments

Calling this function signals an exception to PL/SQL. After a successful return from this function, the external procedure must start its exit handling and return to PL/SQL. Once an exception is signaled to PL/SQL, IN/OUT and OUT arguments, if any, are not processed at all.

### Returns

This function returns `OCIEXTPROC_SUCCESS` if the call succeeds. It returns `OCIEXTPROC_ERROR` if the call fails.

### Related Topics

- OCIExtProcRaiseExcpWithMsg()
  Raises an exception with a message.

## 29.2.4 OCIExtProcRaiseExcpWithMsg()

Raises an exception with a message.

### Purpose

Raises 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.

`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 follows:

```
OCIExtProcContext *with_context;
```

**errnum (IN)**
Oracle Database 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 `errnum`.

**msglen (IN)**
The length of the error message. Pass zero if `errmsg` is a `NULL`-terminated string.

### Comments

This call raises an exception to PL/SQL. In addition, it substitutes the following error message string within the standard Oracle Database error message string.

### Returns

This function returns `OCIEXTPROC_SUCCESS` if the call succeeds. It returns `OCIEXTPROC_ERROR` if the call fails.

**Related Topics**

- OCIExtProcRaiseExcp()
  Raises an Exception to PL/SQL.

## 29.3 Cartridge Services — Memory Services

Lists and describes cartridge services memory services functions.

Table 29-2 lists the memory services functions that are described in this section.

**Table 29-2 Memory Services Functions**

| Function | Purpose |
|---|---|
| OCIDurationBegin() | Start a user duration |
| OCIDurationEnd() | Terminate a user duration |
| OCIMemoryAlloc() | Allocate memory of a given size from a given duration |
| OCIMemoryFree() | Free a memory chunk |
| OCIMemoryResize() | Resize a memory chunk |

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information about using these functions

## 29.3.1 OCIDurationBegin()

Starts a user duration.

### 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. This should be passed as `NULL` for cartridge services.

**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.

**parent (IN)**
The duration number of the parent duration. It is one of the following:

- 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 that uniquely identifies the duration created by this call.

Note that the environment and service context parameters cannot both be `NULL`.

### Related Topics

- OCIDurationEnd()
  Terminates a user duration for a temporary LOB.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.3.2 OCIDurationEnd()

Terminates a user duration.

### Purpose

Terminates a user duration.

### Syntax

```
sword OCIDurationEnd ( OCIEnv             *env,
                       OCIError           *err,
                       const OCISvcCtx    *svc,
                       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)**
OCI service context (this should be passed as `NULL` for cartridge services; otherwise, it should be non-`NULL`).

**duration (IN)**
A user duration previously created by `OCIDurationBegin()`.

### Comments

This function terminates a user duration.

Note that the environment and service context parameters cannot both be `NULL`.

## 29.3.3 OCIMemoryAlloc()

Allocates memory of a given size from a given duration.

### Purpose

Allocates memory of a given size from a given duration.

### Syntax

```
sword OCIMemoryAlloc( void         *hndl,
                      OCIError     *err,
                      void        **mem,
                      OCIDuration   dur,
                      ub4           size,
                      ub4           flags );
```

### Parameters

**hndl (IN)**
The OCI environment handle (`OCIEnv *`) if `dur` is `OCI_DURATION_PROCESS`; otherwise, the user session handle (`OCISession *`).

**err (IN)**
The error handle.

**mem (OUT)**
Memory allocated.

**dur (IN)**
A previously created user duration or one of these values:
`OCI_DURATION_CALLOUT`
`OCI_DURATION_STATEMENT`
`OCI_DURATION_SESSION`
`OCI_DURATION_PROCESS`

**size (IN)**
Size of memory to be allocated.

**flags (IN)**
Set the `OCI_MEMORY_CLEARED` bit to get memory that has been cleared.

### Comments

To allocate memory for the duration of the callout of the agent, that is, external procedure duration, use `OCIExtProcAllocCallMemory()` or `OCIMemoryAlloc()` with `dur` as `OCI_DURATION_CALLOUT`.

### Returns

Error code.

### Related Topics

- OCIExtProcAllocCallMemory()
  Allocates *N* bytes of memory for the duration of the external procedure.

## 29.3.4 OCIMemoryFree()

Frees a memory chunk.

### Purpose

Frees a memory chunk.

### Syntax

```
sword OCIMemoryFree ( void     *hndl,
                      OCIError *err,
                      void     *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.

**Related Topics**

- OCIMemoryAlloc()
  Allocates memory of a given size from a given duration.

## 29.3.5 OCIMemoryResize()

Resizes a memory chunk to a new size.

**Purpose**

Resizes a memory chunk to a new size.

**Syntax**

```
sword OCIMemoryResize( void          *hndl,
                       OCIError      *err,
                       void         **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 the `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.

**Related Topics**

- OCIMemoryAlloc()
  Allocates memory of a given size from a given duration.

## 29.4 Cartridge Services — Maintaining Context

Lists and describes cartridge services maintaining context functions.

Table 29-3 lists the maintaining context functions that are described in this section.

**Table 29-3 Maintaining Context Functions**

| Function | Purpose |
|---|---|
| OCIContextClearValue() | Remove the value stored in the context |
| OCIContextGenerateKey() | Return a unique 4-byte value each time it is called |
| OCIContextGetValue() | Return the value stored in the context |
| OCIContextSetValue() | Save a value (or address) for a particular duration |

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information about using these functions

### 29.4.1 OCIContextClearValue()

Removes the value that is stored in the context associated with the given key (by calling `OCIContextSetValue()`).

**Purpose**

Removes the value that is stored in the context associated with the given key (by calling `OCIContextSetValue()`).

**Syntax**

```
sword OCIContextClearValue( void     *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 value is 255 bytes or `UB1MAXVAL` - 1.

### Comments

An error is returned when a nonexistent key is passed.

### Returns

- If the operation succeeds, the function returns `OCI_SUCCESS`.
- If the operation fails, the function returns `OCI_ERROR`.

## 29.4.2 OCIContextGenerateKey()

Returns a unique, 4-byte value each time it is called.

### Purpose

Returns a unique, 4-byte value each time it is called.

### Syntax

```
sword OCIContextGenerateKey( void     *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.

### Comments

This value is unique for each session.

### Returns

- If the operation succeeds, the function returns `OCI_SUCCESS`.
- If the operation fails, the function returns `OCI_ERROR`.

## 29.4.3 OCIContextGetValue()

Returns the value that is stored in the context associated with the given key (by calling `OCIContextSetValue()`).

### Purpose

Returns the value that is stored in the context associated with the given key (by calling `OCIContextSetValue()`).

### Syntax

```
sword OCIContextGetValue( void       *hndl,
                          OCIError   *err,
                          ub1        *key,
                          ub1         keylen,
                          void      **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 value is 255 bytes or `UB1MAXVAL` - 1.

**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 the operation succeeds, the function returns `OCI_SUCCESS`.
- If the operation fails, the function returns `OCI_ERROR`.

## 29.4.4 OCIContextSetValue()

Saves a value (or address) for a particular duration.

### Purpose

Saves a value (or address) for a particular duration.

### Syntax

```
sword OCIContextSetValue( void        *hndl,
                          OCIError    *err,
                          OCIDuration  duration,
                          ub1         *key,
                          ub1          keylen,
                          void        *ctx_value );
```

### Parameters

**hndl (IN)**
The OCI environment or user session handle.

**err (IN)**
The error handle.

**duration (IN)**
One of these values (a previously created user duration):
`OCI_DURATION_STATEMENT`
`OCI_DURATION_SESSION`

**key (IN)**
Unique key value.

**keylen (IN)**
Length of the key. Maximum value is 255 bytes or `UB1MAXVAL` - 1.

**ctx_value (IN)**
Pointer that is 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 results in overwriting the old context value with the new one. Typically, a client allocates a structure, stores its address in the context using this call, and gets this address in a separate call using `OCIContextGetValue()`. The (key, value) association can be explicitly removed by calling `OCIContextClearValue()`, or else it goes away at the end of the duration.

### Returns

- If the operation succeeds, the function returns `OCI_SUCCESS`.
- If the operation fails, the function returns `OCI_ERROR`.

### Related Topics

- OCIContextClearValue()
  Removes the value that is stored in the context associated with the given key (by calling `OCIContextSetValue()`).

# 29.5 Cartridge Services — Parameter Manager Interface

Lists and describes cartridge services parameter manager interface functions.

Table 29-4 lists the parameter manager interface functions that are described in this section.

**Table 29-4 Parameter Manager Interface Functions**

| Function | Purpose |
|---|---|
| OCIExtractFromFile() | Process the keys and their values in the given file |
| OCIExtractFromList() | Generate a list of values for the parameter denoted by `index` in the parameter list |
| OCIExtractFromStr() | Process the keys and the their values in the given string |
| OCIExtractInit() | Initialize the parameter manager |
| OCIExtractReset() | Reinitialize memory |
| OCIExtractSetKey() | Register information about a key with the parameter manager |
| OCIExtractSetNumKeys() | Inform the parameter manager of the number of keys that are to be registered |
| OCIExtractTerm() | Release all dynamically allocated storage |
| OCIExtractToBool() | Get the Boolean value for the specified key |
| OCIExtractToInt() | Get the integer value for the specified key |
| OCIExtractToList() | Generate a list of parameters from the parameter structures that are stored in memory |
| OCIExtractToOCINum() | Get the number value for the specified key |
| OCIExtractToStr() | Get the string value for the specified key |

**See Also:**

*Oracle Database Data Cartridge Developer's Guide* for more information about using these functions

## 29.5.1 OCIExtractFromFile()

Processes the keys and their values in the given file.

**Purpose**

Processes the keys and their values in the given file.

### Syntax

```
sword OCIExtractFromFile( void     *hndl,
                          OCIError *err,
                          ub4       flag,
                          OraText  *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 file name string.

### Comments

`OCIExtractSetNumKeys()` and `OCIExtractSetKey()` functions must be called to define all of the keys before this routine is called.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

### Related Topics

- OCIExtractSetNumKeys()
  Informs the parameter manager of the number of keys that are to be registered.
- OCIExtractSetKey()
  Registers information about a key with the parameter manager.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.2 OCIExtractFromList()

Generates a list of values for the parameter denoted by index in the parameter list.

### Purpose

Generates a list of values for the parameter denoted by index in the parameter list.

### Syntax

```
sword OCIExtractFromList( void        *hndl,
                          OCIError    *err,
                          uword        index,
                          OraText    **name,
                          ub1         *type,
                          uword       *numvals,
                          void      ***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`; or `OCI_ERROR`.

### Related Topics

- OCIExtractToList()
  Generates a list of parameters from the parameter structures that are stored in memory.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.3 OCIExtractFromStr()

Processes the keys and their values in the given string.

### Purpose

Processes the keys and their values in the given string.

### Syntax

```
sword OCIExtractFromStr( void     *hndl,
                         OCIError *err,
                         ub4       flag,
                         OraText  *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 this routine is called.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

### Related Topics

- OCIExtractSetNumKeys()
  Informs the parameter manager of the number of keys that are to be registered.
- OCIExtractSetKey()
  Registers information about a key with the parameter manager.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.4 OCIExtractInit()

Initializes the parameter manager.

### Purpose

Initializes the parameter manager.

### Syntax

```
sword OCIExtractInit( void      *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 be called only 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`; or `OCI_ERROR`.

#### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.5 OCIExtractReset()

Frees the memory currently used for parameter storage, key definition storage, and parameter value lists and reinitializes the structure.

#### Purpose

Frees the memory currently used for parameter storage, key definition storage, and parameter value lists and reinitializes the structure.

#### Syntax

```
sword OCIExtractReset( void      *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`; or `OCI_ERROR`,

#### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.6 OCIExtractSetKey()

Registers information about a key with the parameter manager.

### Purpose

Registers information about a key with the parameter manager.

### Syntax

```
sword OCIExtractSetKey( void        *hndl,
                        OCIError    *err,
                        const text  *name,
                        ub1          type,
                        ub4          flag,
                        const void  *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 can 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; can 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`). Can 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 `OCIExtractSetNumKeys()` and before calling `OCIExtractFromFile()` or `OCIExtractFromStr()`.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

### Related Topics

- OCIExtractSetNumKeys()
  Informs the parameter manager of the number of keys that are to be registered.
- OCIExtractFromFile()
  Processes the keys and their values in the given file.
- OCIExtractFromStr()
  Processes the keys and their values in the given string.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.7 OCIExtractSetNumKeys()

Informs the parameter manager of the number of keys that are to be registered.

### Purpose

Informs the parameter manager of the number of keys that are to be registered.

### Syntax

```
sword OCIExtractSetNumKeys( void    *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 are to be registered with `OCIExtractSetKey()`.

### Comments

This routine must be called prior to the first call of `OCIExtractSetKey()`.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

**Related Topics**

- OCIExtractSetKey()
  Registers information about a key with the parameter manager.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.8 OCIExtractTerm()

Releases all dynamically allocated storage.

### Purpose

Releases all dynamically allocated storage.

### Syntax

```
sword OCIExtractTerm( void      *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 be called only once.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

**Related Topics**

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.9 OCIExtractToBool()

Gets the Boolean value for the specified key.

### Purpose

The `valno`'th value (starting with 0) is returned.

### Syntax

```
sword OCIExtractToBool( void      *hndl,
                        OCIError  *err,
                        OraText   *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`; or `OCI_ERROR`.

`OCI_NO_DATA` means that there is no `valno`'th value for this key.

#### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.10 OCIExtractToInt()

Gets the integer value for the specified key.

#### Purpose

The `valno`'th value (starting with 0) is returned.

#### Syntax

```
sword OCIExtractToInt( void       *hndl,
                       OCIError   *err,
                       OraText    *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`; or `OCI_ERROR`.

`OCI_NO_DATA` means that there is no `valno`'th value for this key.

#### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.11 OCIExtractToList()

Generates a list of parameters from the parameter structures that are stored in memory.

#### Purpose

Must be called before `OCIExtractValues()` is called.

#### Syntax

```
sword OCIExtractToList( void      *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`; or `OCI_ERROR`.

#### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.12 OCIExtractToOCINum()

Gets the OCINumber value for the specified key.

### Purpose

The `valno`'th value (starting with 0) is returned.

### Syntax

```
sword OCIExtractToOCINum( void      *hndl,
                          OCIError  *err,
                          OraText   *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.

**Related Topics**

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.5.13 OCIExtractToStr()

Gets the string value for the specified key.

### Purpose

The `valno`'th value (starting with 0) is returned.

### Syntax

```
sword OCIExtractToStr( void       *hndl,
                       OCIError   *err,
                       OraText    *keyname,
uword       valno,
OraText    *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`; or `OCI_ERROR`.

`OCI_NO_DATA` means that there is no `valno`'th value for this key.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6 Cartridge Services — File I/O Interface

Lists and describes cartridge services file I/O interface functions.

Table 29-5 lists the file I/O interface functions that are described in this section.

**Table 29-5 File I/O Interface Functions**

| Function | Purpose |
|---|---|
| OCIFileClose() | Close a previously opened file |
| OCIFileExists() | Test to see if the file exists |
| OCIFileFlush() | Write buffered data to a file |
| OCIFileGetLength() | Get the length of a file |
| OCIFileInit() | Initialize the OCIFile package |
| OCIFileOpen() | Open a file |
| OCIFileRead() | Read from a file into a buffer |
| OCIFileSeek() | Change the current position in a file |

**Table 29-5 (Cont.) File I/O Interface Functions**

| Function | Purpose |
|---|---|
| OCIFileTerm() | Terminate the OCIFile package |
| OCIFileWrite() | Write `buflen` bytes into the file |

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information about using these functions

**OCIFileObject**

The `OCIFileObject` data structure holds information about the way in which a file should be opened and the way in which it is accessed after 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().

## 29.6.1 OCIFileClose()

Closes a previously opened file.

### Purpose

Closes a previously opened file.

### Syntax

```
sword OCIFileClose( void          *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 function returns OCI_SUCCESS, the `OCIFileObject` structure pointed to by `filep` is destroyed. Therefore, you should not attempt to access this structure after this function returns OCI_SUCCESS.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6.2 OCIFileExists()

Tests to see if the file exists.

### Purpose

Tests to see if the file exists.

### Syntax

```
sword OCIFileExists( void     *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`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6.3 OCIFileFlush()

Writes buffered data to a file.

### Purpose

Writes buffered data to a file.

### Syntax

```
sword OCIFileFlush( void          *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 file identifier that uniquely references the file.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6.4 OCIFileGetLength()

Gets the length of a file.

### Purpose

Gets the length of a file.

### Syntax

```
sword OCIFileGetLength( void     *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`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6.5 OCIFileInit()

Initializes the OCIFile package.

### Purpose

This function must be called before any other OCIFile routine is called.

### Syntax

```
sword OCIFileInit( void     *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`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6.6 OCIFileOpen()

Opens a file.

### Purpose

Opens a file.

### Syntax

```
sword OCIFileOpen( void           *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 is to be created if it does not exist. Valid values are:
`OCI_FILE_TRUNCATE` — Create a file regardless of whether it exists. If the file exists, overwrite the existing file.
`OCI_FILE_EXCL` — Fail if the file exists; otherwise, create a file.
`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 used with the logical operator OR 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`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6.7 OCIFileRead()

Reads from a file into a buffer.

### Purpose

Reads from a file into a buffer.

### Syntax

```
sword OCIFileRead( void          *hndl,
                   OCIError      *err,
                   OCIFileObject *filep,
                   void          *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 is 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 are read into the user buffer. The read ends either when the user buffer is full, or when it reaches end-of-file.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6.8 OCIFileSeek()

Changes the current position in a file.

### Purpose

Changes the current position in a file.

### Syntax

```
sword OCIFileSeek( void          *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 from which to seek. Use one of the following values:
`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 where reading begins.

**dir (IN)**
The direction to go from the origin.

> **Note:**
>
> The direction can be either `OCIFILE_FORWARD` or `OCIFILE_BACKWARD`.

### Comments

This function allows a seek past the end of the file. Reading from such a position causes an end-of-file condition to be reported. Writing to such a position does 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`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6.9 OCIFileTerm()

Terminates the OCIFile package.

### Purpose

This function must be called after the OCIFile package is no longer being used.

### Syntax

```
sword OCIFileTerm( void     *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`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.6.10 OCIFileWrite()

Writes `buflen` bytes into the file.

### Purpose

Writes `buflen` bytes into the file.

### Syntax

```
sword OCIFileWrite( void          *hndl,
                    OCIError      *err,
                    OCIFileObject *filep,
                    void          *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 which the data is written. The length of the allocated memory is assumed to be `buflen`.

**buflen (IN)**
The length of the buffer in bytes.

**byteswritten (OUT)**
The number of bytes written.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.7 Cartridge Services — String Formatting Interface

Lists and describes cartridge services string formatting functions.

Table 29-6 lists the string formatting functions that are described in this section.

**Table 29-6 String Formatting Functions**

| Function | Purpose |
|---|---|
| OCIFormatInit() | Initialize the OCIFormat package |
| OCIFormatString() | Write a text string into the supplied text buffer |
| OCIFormatTerm() | Terminate the OCIFormat package |

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information about using these functions

## 29.7.1 OCIFormatInit()

Initializes the OCIFormat package.

### Purpose

Initializes the OCIFormat package.

### Syntax

```
sword OCIFormatInit( void      *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 be called only once.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.7.2 OCIFormatString()

Writes a text string into the supplied text buffer using the argument list submitted to it and in accordance with the format string given.

### 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( void          *hndl,
                       OCIError      *err,
                    OraText       *buffer,
                    sbig_ora       bufferLength,
                    sbig_ora      *returnLength,
                    const OraText *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 percent 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 Table 29-7 and Table 29-8.

**... (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.

### Format Modifiers

A format modifier alters or extends the format specification, allowing more specialized output. The format modifiers, as described in Table 29-7, can be in any order and are all optional.

**Table 29-7    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 always has a radix character.

Field Width

`<w>` where `<w>` is a number specifying a minimum field width. The converted argument is 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 is 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 for `<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> (a period followed by the number <p>), specifies 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 zeros are added to make up the difference). The special character `'*'` may be used for <p>, indicating that 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 format codes, as described in Table 29-8, can appear in uppercase, which causes all alphabetic characters in the output to appear in uppercase except for text strings, which are not converted.

**Table 29-8 Format Codes to Specify How to Format an Argument Written to a String**

| 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 fits in the specified precision (default precision if unspecified), and choosing `'e'` only if exponential format allows 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 29-8 (Cont.) Format Codes to Specify How to Format an Argument Written to a String

| Codes | Operation |
|---|---|
| `'s'` | Prints an argument using the default format code for its type:<br>`ociformatub<n>`, `ociformatuword`, `ociformatubig_ora`, `ociformateb<n>`, and `ociformateword`.<br>The format code used is 'u'.<br>`ociformatsb<n>`, `ociformatsword`, and `ociformatsbig_ora`.<br>The format code used is 'd'.<br>`ociformatchar`<br>The format code used is 'c'.<br>`ociformattext`<br>Prints text until trailing `NULL` is found.<br>`ociformatdouble`<br>The format code used is 'g'.<br>`ociformatdvoid`<br>The format code used is 'p'.<br>`' %' - print a '%'.` |
| `'u'` | Unsigned decimal integer |
| 'x' | Unsigned hexadecimal integer |

## Example

### Using OCIFormatString() to Format a Date Two Different Ways for Two Countries

```
/* This example shows the power of arbitrary argument    */
/* selection in the context of internationalization.  A  */
/* date is formatted in two different ways for two 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;
void     *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". */
```

**Returns**

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

**Related Topics**

- OCIFormatInit()
  Initializes the OCIFormat package.
- OCIFormatTerm()
  Terminates the OCIFormat package.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 29.7.3 OCIFormatTerm()

Terminates the OCIFormat package.

**Purpose**

Terminates the OCIFormat package.

**Syntax**

```
sword OCIFormatTerm( void     *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 be called only once.

**Returns**

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

**Related Topics**

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

### 30

# 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 Database installation. For additional information, see 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

## 30.1 Introduction to Any Type and Data Interfaces

This section describes the OCI Any Type and Data functions in detail.

> **See Also:**
>
> "AnyType, AnyData, and AnyDataSet Interfaces"

### 30.1.1 Conventions for OCI Functions

Describes the conventions for OCI functions. The entries for each function may also contain function return values.

See the Conventions for OCI Functions for the conventions used in describing each function.

The OCI Any Type and Data functions typically return one of the values described in the following table.

**Table 30-1 Function Return Values**

| Return Value | Meaning |
|---|---|
| `OCI_SUCCESS` | The operation succeeded. |

**Table 30-1 (Cont.) Function Return Values**

| Return Value | Meaning |
|---|---|
| `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:**
>
> Error Handling in OCI for more information about return codes and error handling

# 30.2 OCI Type Interface Functions

Table 30-2 lists the Type Interface functions that are described in this section.

**Table 30-2 Type Interface Functions**

| Function | Purpose |
|---|---|
| OCITypeAddAttr() | Add an attribute to an object type that was constructed earlier with typecode `OCI_TYPECODE_OBJECT` |
| OCITypeBeginCreate() | Begin the construction process for a transient type. The type is anonymous (no name). |
| OCITypeEndCreate() | Finish construction of a type description. Subsequently, only access is allowed. |
| OCITypeSetBuiltin() | Set 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() | Set collection type information. This call can be made only if the type has been constructed with a collection typecode. |

## 30.2.1 OCITypeAddAttr()

Adds an attribute to an object type that was constructed earlier with typecode `OCI_TYPECODE_OBJECT`.

**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 the attribute name, in bytes.

**attr_info (IN)**
Information about the attribute. It is obtained by allocating an `OCIParam` parameter handle and setting type information in the `OCIParam` using `OCIAttrSet()` calls.

### Related Topics

- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.2.2 OCITypeBeginCreate()

Begins the construction process for a transient type. The type is anonymous (no name).

### Purpose

Begins the construction process for a transient type. The type is 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
- `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 typecodes are specified in "Typecodes". Additional information about 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. It is one of these:

- 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.

### Related Topics

- OCITypeAddAttr()
  Adds an attribute to an object type that was constructed earlier with typecode `OCI_TYPECODE_OBJECT`.
- OCITypeSetCollection()
  Sets collection type information.
- OCITypeEndCreate()
  Finishes construction of a type description. Subsequently, only access is allowed.
- OCITypeSetBuiltin()
  Sets built-in type information.

- OCIDurationBegin()
  Starts a user duration.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.2.3 OCITypeEndCreate()

Finishes construction of a type description. Subsequently, only access is allowed.

### Purpose

Finishes construction of a type description. Subsequently, only access is 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.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.2.4 OCITypeSetBuiltin()

Sets built-in type information.

### Purpose

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 about the built-in type (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.

**Related Topics**

- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.2.5 OCITypeSetCollection()

Sets collection type information.

**Purpose**

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).

**Related Topics**

- OCIAttrSet()
  Sets the value of an attribute of a handle or a descriptor.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3 OCI Any Data Interface Functions

Lists and describes OCI Any Data interface functions.

Table 30-3 lists the Any Data Interface functions that are described in this section.

**Table 30-3 Any Data Functions**

| Function | Purpose |
|---|---|
| OCIAnyDataAccess() | Retrieve the data value of an `OCIAnyData` |
| OCIAnyDataAttrGet() | Get the value of the attribute at the current position in the `OCIAnyData` |
| OCIAnyDataAttrSet() | Set the attribute at the current position with a given value |
| OCIAnyDataBeginCreate() | Allocate an `OCIAnyData` for the given duration and initialize it with the type information |
| OCIAnyDataCollAddElem() | Add the next collection element to the collection attribute of the `OCIAnyData` at the current attribute position |
| OCIAnyDataCollGetElem() | Access sequentially the elements in the collection attribute at the current position in the `OCIAnyData` |
| OCIAnyDataConvert() | Construct an `OCIAnyData` with the given data value of the given type |
| OCIAnyDataDestroy() | Free an AnyData |
| OCIAnyDataEndCreate() | Mark the end of `OCIAnyData` creation |
| OCIAnyDataGetCurrAttrNum() | Return the current attribute number of the `OCIAnyData` |
| OCIAnyDataGetType() | Get the type corresponding to an AnyData value |
| OCIAnyDataIsNull() | Check if `OCIAnyData` is `NULL` |
| OCIAnyDataTypeCodeToSqlt() | Convert the `OCITypeCode` for an AnyData value to the SQLT code that corresponds to the representation of the value as returned by the OCIAnyData API |

## 30.3.1 OCIAnyDataAccess()

Retrieve the data value of an `OCIAnyData`.

### Purpose

The data value should be of the type with which the `OCIAnyData` was initialized. You can use this call 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,
                         void         *null_ind,
                         void         *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 any of the following types, then this parameter should be not `NULL`.

- `OCI_TYPECODE_OBJECT`
- `OCI_TYPECODE_REF`
- `OCI_TYPECODE_VARRAY`
- `OCI_TYPECODE_TABLE`

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 is `OCI_IND_NOTNULL` if the value is not `NULL`, and it is `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 (is 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 does not give the length, in bytes, implicitly.

### Related Topics

- OCIAnyDataAttrGet()
  Gets the value of the attribute at the current position in the `OCIAnyData`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.2 OCIAnyDataAttrGet()

Gets the value of the attribute at the current position in the `OCIAnyData`.

### Purpose

Attribute values can be accessed sequentially.

### Syntax

```
sword OCIAnyDataAttrGet ( OCISvcCtx    *svchp,
                          OCIError     *errhp,
                          OCIAnyData   *sdata,
                          OCITypeCode   tc,
                          OCIType      *attr_type,
                          void         *null_ind,
                          void         *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 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]**
The `attr_type` parameter 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 `(void **)` in `null_ind`.
The indicator returned is `OCI_IND_NOTNULL` if the value is not `NULL`, and it is `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 does not give the length, in bytes, implicitly.

**is_any (IN)**
Is attribute to be returned in the form of `OCIAnyData`?

### Comments

You can use this call with an `OCIAnyData` of typecode `OCI_TYPECODE_OBJECT` only.

- This call gets the value of the attribute at the current position in the `OCIAnyData`.
- The `tc` parameter must match the type of the attribute at the current position; otherwise, an error is returned.
- The `is_any` parameter is applicable only when the typecode of the attribute is one of these values:
  - `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()`. For built-in types such as `NUMBER` and `VARCHAR`, the attribute can be just a pointer to a stack variable. Table 30-4 lists the available Oracle data types that can be used as object attribute types and the corresponding types of the attribute value that should be passed.

**Table 30-4    Data Types and Attribute Values**

| Data Types | attr_value |
|---|---|
| `VARCHAR2, VARCHAR, CHAR` | `OCIString **` |
| `NUMBER, REAL, INT, FLOAT, DECIMAL` | `OCINumber **` |
| `DATE` | `OCIDate **` |

**Table 30-4 (Cont.) Data Types and Attribute Values**

| Data Types | attr_value |
|---|---|
| `TIMESTAMP` | `OCIDateTime **` |
| `TIMESTAMP WITH TIME ZONE` | `OCIDateTime **` |
| `TIMESTAMP WITH LOCAL TIME ZONE` | `OCIDateTime **` |
| `INTERVAL YEAR TO MONTH` | `OCIInterval **` |
| `INTERVAL DAY TO SECOND` | `OCIInterval **` |
| `BLOB` | `OCILobLocator **` or `OCIBlobLocator **` |
| `CLOB` | `OCILobLocator **` or `OCIClobLocator *` |
| `BFILE` | `OCILobLocator **` |
| `REF` | `OCIRef **` |
| `RAW` | `OCIRaw **` |
| `VARRAY` | `OCIArray **` (or `OCIAnyData *` if `is_any` is `TRUE`) |
| `TABLE` | `OCITable **` (or `OCIAnyData *` if `is_any` is `TRUE`) |
| `OBJECT` | `void **` (or `OCIAnyData *` if `is_any` is `TRUE`) |

**Related Topics**

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.3 OCIAnyDataAttrSet()

Sets the attribute at the current position with a given value.

### 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,
                          void         *null_ind,
                          void         *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]**
The `attr_type` parameter gives the type description of the referenced type (for `OCI_TYPECODE_REF`), the type description of the collection type (for `OCI_TYPECODE_VARRAY`, `OCI_TYPECODE_TABLE`), and 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 does 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 `OCIAnyDataCollAddElem()` (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 the state maintained inside the `OCIAnyData`. Piece-wise construction of embedded attributes and collection elements is 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.

The `tc` parameter 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.

Table 30-5 lists the available data types that can be used as object attribute types and the corresponding types of the attribute value that should be passed.

**Table 30-5 Data Types and Attribute Values**

| Data Types | 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` | `void *` (or `OCIAnyData *` if `is_any` is `TRUE`) |

**Related Topics**

- OCIAnyDataBeginCreate()
  Allocates an `OCIAnyData` for the given duration and initializes it with the type information.
- OCIAnyDataCollAddElem()
  Adds the next collection element to the collection attribute of the `OCIAnyData` at the current attribute position.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.4 OCIAnyDataBeginCreate()

Allocates an `OCIAnyData` for the given duration and initializes it with the type information.

### 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()`.

**tc (IN)**
Typecode corresponding to `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. It is one of these:

- A user duration that was previously created. It can be created by using .
- 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 `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 `OCIAnyDataCollAddElem()` for the collection typecodes.

Attribute values must be set in order. They must be set 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 is not yet supported.

For performance reasons, `OCIAnyData` ends up pointing to the `OCIType` parameter passed in. You must ensure that the `OCIType` lives longer (has an allocation duration >= the duration of `OCIAnyData`, if the `OCIType` is a transient one, or has an allocation or pin duration >= the duration of `OCIAnyData`, if the `OCIType` is a persistent one).

**Related Topics**

- OCIDurationBegin()
  Starts a user duration.
- OCIAnyDataAttrSet()
  Sets the attribute at the current position with a given value.
- OCIAnyDataCollAddElem()
  Adds the next collection element to the collection attribute of the `OCIAnyData` at the current attribute position.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.5 OCIAnyDataCollAddElem()

Adds the next collection element to the collection attribute of the `OCIAnyData` at the current attribute position.

### Purpose

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,
                              void         *null_ind,
                              void         *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]**
The `collelem_type` parameter gives the type description of the referenced type (for `OCI_TYPECODE_REF`), the type description of the collection type (for `OCI_TYPECODE_NAMEDCOLLECTION`), and 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 are ignored. Otherwise, `null_ind` must be set to `OCI_IND_NOTNULL`.

See `OCIAnyDataAttrSet()` for the type of attribute to be passed in for all the possible types of the collection elements.

### Related Topics

- OCIAnyDataAttrSet()
  Sets the attribute at the current position with a given value.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.6 OCIAnyDataCollGetElem()

Accesses sequentially the elements in the collection attribute at the current position in the `OCIAnyData`.

### 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,
                              void         *null_ind,
                              void         *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]**
The `collelem_type` parameter gives the type description of the referenced type (for `OCI_TYPECODE_REF`), the type description of the collection type (for `OCI_TYPECODE_NAMEDCOLLECTION`), and 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 `(void **)` 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 the appropriate type. Otherwise, an error is returned.

As for `OCIAnyDataAttrGet()`, the `is_any` parameter is applicable only if the `collelem_tc` typecode is `OCI_TYPECODE_OBJECT`. If `is_any` is `TRUE`, the `attr_value` is in the form of `OCIAnyData *`.

This call returns `OCI_NO_DATA` when the end of the collection has been reached. It returns `OCI_SUCCESS` upon success and `OCI_ERROR` upon error.

See `OCIAnyDataAttrGet()` for the type of attribute to be passed in for all the possible types of the collection elements.

### Related Topics

- OCIAnyDataAttrGet()
  Gets the value of the attribute at the current position in the `OCIAnyData`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.7 OCIAnyDataConvert()

Constructs an `OCIAnyData` with the given data value that is of the given type.

### Purpose

You can use this call 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,
                          void         *null_ind,
                          void         *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`, or `OCI_TYPECODE_VARRAY`).
If (`*sdata`) is not `NULL` and it represents a skeleton instance returned during the `OCIAnyDataSetAddInstance()`, the `tc` and the `inst_type` parameters are optional here. This is because the type information for such a skeleton instance is known. If the `tc` and `inst_type` parameters are provided for this situation, they are 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. It is one of these:

- 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 is `OCI_IND_NOTNULL` if the value is not `NULL`, and it is `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 does not give the length implicitly.

**sdata (IN/OUT)**
Initialized `OCIAnyData`. If (`*sdata`) is not `NULL` at the beginning of the call, the memory could be reused instead of reallocating space for `OCIAnyData`.
Therefore, do not pass an uninitialized pointer here.
If (`*sdata`) represents a skeleton instance returned during an `OCIAnyDataSetAddInstance()` call, the `tc` and `inst_type` parameters are used for type checking, if necessary.

### Comments

For performance reasons, `OCIAnyData` pointer ends up pointing to the passed in `OCIType` parameter. You must ensure that the `OCIType` lives longer (has an allocation duration >= the duration of `OCIAnyData`, if the `OCIType` is a transient one, or has an allocation or pin duration >= the duration of `OCIAnyData`, if the `OCIType` is a persistent one).

**Related Topics**

- OCIAnyDataSetAddInstance()
  Adds a new skeleton instance to the `OCIAnyDataSet` and sets all the attributes of the instance to `NULL`.
- OCIAnyDataAttrSet()
  Sets the attribute at the current position with a given value.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.8 OCIAnyDataDestroy()

Free an OCIAnyData.

### Purpose

Frees an `OCIAnyData`.

### Syntax

```
sword OCIAnyDataDestroy ( OCISvcCtx      *svchp,
                          OCIError       *errhp,
                          OCIAnyData     *sdata );
```

### Parameters

**svchp (IN)**
The OCI service context.

**errhp (IN/OUT)**
The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

**sdata (IN/OUT)**
Pointer to a type of `OCIAnyData` to be freed.

**Related Topics**

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.9 OCIAnyDataEndCreate()

Marks the end of `OCIAnyData` creation.

### Purpose

It should be called after initializing all attributes of its instances with suitable values. This call is valid only if `OCIAnyDataBeginCreate()` was 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`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.10 OCIAnyDataGetCurrAttrNum()

Returns the current attribute number of `OCIAnyData`.

### Purpose

If `OCIAnyData` is being constructed, this function refers to the current attribute that is being set. Otherwise, if `OCIAnyData` is being accessed, this function 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.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.11 OCIAnyDataGetType()

Gets the type corresponding to an `OCIAnyData` value.

### Purpose

It returns the actual pointer to the type maintained inside an `OCIAnyData`. No copying is done for performance reasons. Do not use this type after 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 is `NULL` if the `OCIAnyData` corresponds to a built-in type.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.12 OCIAnyDataIsNull()

Checks if the content of the type within the `OCIAnyData` is `NULL`.

### Purpose

Checks if the content 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`; otherwise, `FALSE`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.3.13 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.

### Purpose

Converts the `OCITypeCode` for an `OCIAnyData` 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` and `OCI_TYPECODE_VARCHAR2` to `SQLT_VST` (which corresponds to the `OCIString` mapping) with a charset form of `SQLCS_IMPLICIT`. `OCI_TYPECODE_NVARCHAR2` also returns `SQLT_VST` (`OCIString` mapping is used by the OCIAnyData API) with a charset form of `SQLCS_NCHAR`.

**Related Topics**

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- NCHAR Typecodes for OCIAnyData Functions
  The function `OCIAnyDataTypeCodeToSqlt()` converts the `OCITypeCode` for an OCIAnyData value to the SQLT code that corresponds to the representation of the value as returned by the OCIAnyData API.

## 30.4 OCI Any Data Set Interface Functions

Lists and describes OCI Any Data set interface functions.

Table 30-6 lists the Any Data Set Interface functions that are described in this section.

**Table 30-6 Any Data Set Functions**

| Function | Purpose |
|---|---|
| OCIAnyDataSetAddInstance() | Add a new skeleton instance to the `OCIAnyDataSet` and set all the attributes of the instance to `NULL` |
| OCIAnyDataSetBeginCreate() | Allocate an `OCIAnyDataSet` for the given duration and initialize it with the type information |
| OCIAnyDataSetDestroy() | Free the `OCIAnyDataSet` |
| OCIAnyDataSetEndCreate() | Mark the end of `OCIAnyDataSet` creation |
| OCIAnyDataSetGetCount() | Get the number of instances in the `OCIAnyDataSet` |
| OCIAnyDataSetGetInstance() | Return the `OCIAnyData` corresponding to an instance at the current position and update the current position |
| OCIAnyDataSetGetType() | Get the type corresponding to an `OCIAnyDataSet` |

## 30.4.1 OCIAnyDataSetAddInstance()

Adds a new skeleton instance to the `OCIAnyDataSet` and sets all the attributes of the instance to `NULL`.

### Purpose

Adds a new skeleton instance to the `OCIAnyDataSet` and sets all the attributes of the instance 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` is allocated for the same duration as the `OCIAnyDataSet`. If `(*data)` is not `NULL`, it is 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 that can be constructed subsequently by invoking the OCIAnyData API.

> **Note:**
>
> The old value is not destroyed. You must 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`.

**Related Topics**

- OCIAnyDataConvert()
  Constructs an `OCIAnyData` with the given data value that is of the given type.
- OCIAnyDataAttrSet()
  Sets the attribute at the current position with a given value.
- OCIAnyDataCollAddElem()
  Adds the next collection element to the collection attribute of the `OCIAnyData` at the current attribute position.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.4.2 OCIAnyDataSetBeginCreate()

Allocates an `OCIAnyDataSet` for the given duration and initializes it with the type information.

### Purpose

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, such as `OCI_TYPECODE_NUMBER`, this parameter can be `NULL`. It should be non-`NULL` for user-defined types, such as `OCI_TYPECODE_OBJECT`, `OCI_TYPECODE_REF`, and collection types.

**dur (IN)**
Duration for which `OCIAnyDataSet` is allocated. It is one of these:

- 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` ends up pointing to the `OCIType` parameter passed in. You must ensure that the `OCIType` lives longer (has an allocation duration >= the duration of the `OCIAnyData` if the `OCIType` is a transient one, or has allocation or pin duration >= the duration of the `OCIAnyData`, if the `OCIType` is a persistent one).

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.4.3 OCIAnyDataSetDestroy()

Frees the `OCIAnyDataSet`.

### 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 `OCI_ERROR`.

**data_set (IN/OUT)**
`OCIAnyDataSet` to be freed.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.4.4 OCIAnyDataSetEndCreate()

Marks the end of `OCIAnyDataSet` creation.

### Purpose

This function 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 `OCI_ERROR`.

**data_set (IN/OUT)**
Initialized `OCIAnyDataSet`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.4.5 OCIAnyDataSetGetCount()

Gets the number of instances in the `OCIAnyDataSet`.

### 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 `OCI_ERROR`.

**data_set (IN/OUT)**
A well-formed `OCIAnyDataSet`.

**count (OUT)**
Number of instances in `OCIAnyDataSet`.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.4.6 OCIAnyDataSetGetInstance()

Returns the `OCIAnyData` corresponding to an instance at the current position and updates the current position.

### 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 `OCI_ERROR`.

**data_set (IN/OUT)**
A well-formed `OCIAnyDataSet`.

**data (IN/OUT)**
`OCIAnyData` corresponding to the instance. If (*data) is `NULL`, a new `OCIAnyData` is allocated for same duration as the `OCIAnyDataSet`. If (`*data`) is not `NULL`, it is 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 can be used to access the instance.

**Related Topics**

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 30.4.7 OCIAnyDataSetGetType()

Gets the type corresponding to an `OCIAnyDataSet`.

### 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 `OCI_ERROR`.

**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 is `NULL` if the `OCIAnyData` corresponds to a built-in type.

### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

# 31

# OCI Globalization Support Functions

This chapter describes the OCI globalization support functions.

This chapter contains these topics:

- Introduction to Globalization Support in OCI
- OCI Locale Functions
- OCI Locale-Mapping Function
- OCI String Manipulation Functions
- OCI Character Classification Functions
- OCI Character Set Conversion Functions
- OCI Messaging Functions

## 31.1 Introduction to Globalization Support in OCI

This section describes the globalization support functions in detail.

> **See Also:**
>
> *Oracle Database Globalization Support Guide*

### 31.1.1 Conventions for OCI Functions

Describes the conventions for OCI functions.

The entries for each function may also contain the following information:

- Returns — The values returned. The standard return values have the following meanings as described in the following table.

**Table 31-1 Function Return Values**

| Return Value | Meaning |
|---|---|
| `OCI_SUCCESS` | The operation succeeded. |
| `OCI_ERROR` | The operation failed. The specific error can be retrieved by calling `OCIErrorGet()` on the error handle passed to the function. |
| `OCI_INVALID_HANDLE` | The OCI handle passed to the function is invalid. |

**See Also:**

- Conventions for OCI Functionsfor the conventions used in describing each function
- Error Handling in OCI for more information about return codes and error handling

# 31.2 OCI Locale Functions

Lists and describes OCI Locale Functions.

Table 31-2 lists the OCI locale functions that are described in this section.

An Oracle locale consists of language, territory, and character set definitions. The locale determines conventions such as day and month names, and 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 31-2 OCI Locale Functions**

| Function | Purpose |
|---|---|
| OCINlsCharSetIdToName() | Return the Oracle Database character set name from the specified character set ID |
| OCINlsCharSetNameToId() | Return the Oracle Database character set ID for the specified Oracle Database character set name |
| OCINlsEnvironmentVariableGet() | Return the character set ID from `NLS_LANG` or the national character set ID from `NLS_NCHAR` |
| OCINlsGetInfo() | Copy locale information from an OCI environment or user session handle into an array pointed to by the destination buffer within a specified size |
| OCINlsNumericInfoGet() | Copy numeric language information from the OCI environment or user session handle into an output number variable |

## 31.2.1 OCINlsCharSetIdToName()

Returns the Oracle Database character set name from the specified character set ID.

**Purpose**

Returns the Oracle Database character set name from the specified character set ID.

### Syntax

```
sword OCINlsCharSetIdToName ( void      *hndl,
                              OraText   *buf,
                              size_t     buflen
                              ub2        id );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user 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 Database 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 Database character set ID.

### Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

## 31.2.2 OCINlsCharSetNameToId()

Returns the Oracle Database character set ID for the specified Oracle Database character set name.

### Purpose

Returns the Oracle Database character set ID for the specified Oracle Database character set name.

### Syntax

```
ub2 OCINlsCharSetNameToId ( void            *hndl,
                            const OraText   *name );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle. If the handle is invalid, then the function returns zero.

**name (IN)**
Pointer to a `NULL`-terminated Oracle Database 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`.

## 31.2.3 OCINlsEnvironmentVariableGet()

Returns the character set ID from `NLS_LANG` or the national character set ID from `NLS_NCHAR`.

### Purpose

Returns the character set ID from `NLS_LANG` or the national character set ID from `NLS_NCHAR`.

### Syntax

```
sword OCINlsEnvironmentVariableGet ( void        *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. For numeric data, this argument is ignored.

**item (IN)**
Specifies one of these values to get from the globalization support environment variable:

- `OCI_NLS_CHARSET_ID`: `NLS_LANG` character set ID in `ub2` data type; if `NLS_LANG` is not set, the default character set ID is returned (ASCII).
- `OCI_NLS_NCHARSET_ID`: `NLS_NCHAR` character set ID in `ub2` data type; if `NLS_NCHAR` is not set, the value of `OCI_NLS_CHARSET_ID` is returned instead.

**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. For numeric data, this argument is ignored.

**rsize (OUT)**
The length of the return value in bytes.

### Comments

To allow for future enhancements of this function (to retrieve other values from environment variables) the data type of the output `val` is a pointer to `void`. 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_ID` and `OCI_ATTR_ENV_NCHARSET_ID`, respectively.

### Returns

`OCI_SUCCESS`; or `OCI_ERROR`.

### Related Topics

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.2.4 OCINlsGetInfo()

Copy locale information from an OCI environment or user session handle into an array pointed to by the destination buffer within a specified size.

### Purpose

Obtains locale information from an OCI environment or user session handle to an array pointed to by the destination buffer within a specified size.

### Syntax

```
sword OCINlsGetInfo ( void        *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 must 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 these 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 Database 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 Database default date format
`OCI_NLS_DTIMEFORMAT`: Oracle Database 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`.

#### Related Topics

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

### 31.2.5 OCINlsNumericInfoGet()

Copy numeric language information from the OCI environment or user session handle into an output number variable.

#### Purpose

Obtains numeric language information from the OCI environment or user session handle and puts it into an output number variable.

#### Syntax

```
sword OCINlsNumericInfoGet ( void        *hndl,
                             OCIError    *errhp,
                             sb4         *val,
                             ub2          item );
```

**Parameters**

**hndl (IN/OUT)**
The OCI environment or user session handle. If the handle is invalid, it returns `OCI_INVALID_HANDLE`.

**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`.

**Related Topics**

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 31.3 OCI Locale-Mapping Function

Lists and describes OCI locale-mapping functions.

Table 31-3 lists an OCI locale-mapping function that is described in this section.

The OCI locale-mapping function performs name mapping to and from Oracle Database, Internet Assigned Numbers Authority (IANA), and International Organization for Standardization (ISO) names for character set names, language names, and territory names.

**Table 31-3 OCI Locale-Mapping Function**

| Function | Purpose |
|---|---|
| OCINlsNameMap() | Map Oracle Database character set names, language names, and territory names to and from Internet Assigned Numbers Authority (IANA) and International Organization for Standardization (ISO) names |

## 31.3.1 OCINlsNameMap()

Maps Oracle Database character set names, language names, and territory names to and from Internet Assigned Numbers Authority (IANA) and International Organization for Standardization (ISO) names.

### Purpose

Maps Oracle Database 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 ( void               *hndl,
                      OraText            *buf,
                      size_t              buflen,
                      const OraText      *srcbuf,
                      uword               flag );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user 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 Database

`OCI_NLS_CS_ORA_TO_IANA`: Map character set name from Oracle Database to IANA

`OCI_NLS_LANG_ISO_TO_ORA`: Map language name from ISO to Oracle Database

`OCI_NLS_LANG_ORA_TO_ISO`: Map language name from Oracle Database to ISO

`OCI_NLS_LOCALE_A2_ISO_TO_ORA`: Map locale name from A2 ISO to Oracle Database

`OCI_NLS_LOCALE_ORA_TO_A2_ISO`: Map locale name from Oracle Database to A2 ISO

`OCI_NLS_TERR_ISO_TO_ORA`: Map territory name from ISO to Oracle Database

`OCI_NLS_TERR_ORA_TO_ISO`: Map territory name from Oracle Database to ISO

`OCI_NLS_TERR_ISO3_TO_ORA`: Map territory name from 3-letter ISO abbreviation to Oracle Database

`OCI_NLS_TERR_ORA_TO_ISO3`: Map territory name from Oracle Database to 3-letter ISO abbreviation

**Returns**

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`; or `OCI_ERROR`.

## 31.4 OCI String Manipulation Functions

Lists and describes 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 of the string calculated in characters.

The wide-character data type is Oracle-specific and should not be confused with the `wchar_t` data type defined by the ANSI/ISO C standard. The Oracle wide-character data type is always 4 bytes in all operating systems, whereas the size of `wchar_t` depends on the implementation and the operating system. The Oracle wide-character data type 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 31-4 summarizes the OCI string manipulation functions, which are described in this section.

**Table 31-4 OCI String Manipulation Functions**

| Function | Purpose |
|---|---|
| OCIMultiByteInSizeToWideChar() | Convert part of a multibyte string into the wide-character string |
| OCIMultiByteStrCaseConversion() | Convert a multibyte string into the specified case and copies the result into the destination array |
| OCIMultiByteStrcat() | Append a multibyte string to the destination string |

**Table 31-4 (Cont.) OCI String Manipulation Functions**

| Function | Purpose |
|---|---|
| OCIMultiByteStrcmp() | Compare two multibyte strings by binary, linguistic, or case-insensitive comparison methods |
| OCIMultiByteStrcpy() | Copy a multibyte string into the destination array. It returns the number of bytes copied. |
| OCIMultiByteStrlen() | Return the number of bytes in a multibyte string |
| OCIMultiByteStrncat() | Append, at most, *n* bytes from a multibyte string to the destination string |
| OCIMultiByteStrncmp() | Compare two multibyte strings by binary, linguistic, or case-insensitive comparison methods. Each string is in the specified length |
| OCIMultiByteStrncpy() | Copy a specified number of bytes of a multibyte string into the destination array |
| OCIMultiByteStrnDisplayLength() | Return the number of display positions occupied by the multibyte string within the range of *n* bytes |
| OCIMultiByteToWideChar() | Convert a `NULL`-terminated multibyte string into wide-character format |
| "OCIWideCharInSizeToMultiByte()" | Convert part of a wide-character string to the multibyte string |
| OCIWideCharMultiByteLength() | Determine the number of bytes required for a wide character in multibyte encoding |
| OCIWideCharStrCaseConversion() | Convert a wide-character string into the specified case and copies the result into the destination array |
| OCIWideCharStrcat() | Append a wide-character string to the destination string |
| OCIWideCharStrchr() | Search for the first occurrence of a wide character in a string. Return a point to the wide character if the search is successful. |
| OCIWideCharStrcmp() | Compare two wide-character strings by binary, linguistic, or case-insensitive comparison methods |
| OCIWideCharStrcpy() | Copy a wide-character string into a destination array. Return the number of characters copied. |
| OCIWideCharStrlen() | Return the number of characters in a wide-character string |
| OCIWideCharStrncat() | Append, at most, *n* characters from a wide-character string to the destination string |
| OCIWideCharStrncmp() | Compare two wide-character strings by binary, linguistic, or case-insensitive methods. Each string is a specified length. |
| OCIWideCharStrncpy() | Copy, at most, *n* characters of a wide-character string into the destination array |

**Table 31-4 (Cont.) OCI String Manipulation Functions**

| Function | Purpose |
|---|---|
| OCIWideCharStrrchr() | Search for the last occurrence of a character in a wide-character string |
| OCIWideCharToLower() | Convert a specified wide character into the corresponding lowercase character |
| OCIWideCharToMultiByte() | Convert a `NULL`-terminated wide-character string into a multibyte string |
| OCIWideCharToUpper() | Convert a specified wide character into the corresponding uppercase character |

## 31.4.1 OCIMultiByteInSizeToWideChar()

Converts part of a multibyte string into the wide-character string.

### Purpose

Converts part of a multibyte string into the wide-character string.

### Syntax

```
sword OCIMultiByteInSizeToWideChar ( void          *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 a `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 the destination buffer, or number of characters for the 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 Topics**

- OCIMultiByteToWideChar()
  Converts a `NULL`-terminated multibyte string into wide-character format.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.2 OCIMultiByteStrCaseConversion()

Converts a multibyte string into the specified case and copies the result into the destination array.

**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 ( void           *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 in the result string, not including the `NULL` terminator.

## 31.4.3 OCIMultiByteStrcat()

Appends a multibyte string to the destination string.

#### 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 ( void            *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 Topics

- OCIMultiByteStrncat()
  Appends, at most, *n* bytes from a multibyte string to the destination string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.4 OCIMultiByteStrcmp()

Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods.

### Purpose

Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods.

### Syntax

```
int OCIMultiByteStrcmp ( void            *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 these 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

The return values for the function are:

- 0, if `str1 = str2`
- Positive, if `str1 > str2`
- Negative, if `str1 < str2`

### Related Topics

- OCIMultiByteStrncmp()
  Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods. Each string is in the specified length.

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.5 OCIMultiByteStrcpy()

Copies a multibyte string into the destination array. It returns the number of bytes copied.

### Purpose

Copies the multibyte string pointed to by `srcstr` into the array pointed to by `dststr`.

### Syntax

```
size_t OCIMultiByteStrcpy ( void            *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 Topics

- OCIMultiByteStrncpy()
  Copies a specified number of bytes of a multibyte string into the destination array.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.6 OCIMultiByteStrlen()

Returns the number of bytes in a multibyte string.

### Purpose

Returns the number of bytes in the multibyte string pointed to by `str`, not including the `NULL` terminator.

### Syntax

```
size_t OCIMultiByteStrlen ( void           *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 Topics

- OCIWideCharStrlen()
  Return the number of characters in a wide-character string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.7 OCIMultiByteStrncat()

Appends, at most, *n* bytes from a multibyte string to the destination string.

### Purpose

Appends a specified number of bytes from a multibyte string to a destination string.

### Syntax

```
size_t OCIMultiByteStrncat ( void           *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. The `dststr` parameter 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 Topics

- OCIMultiByteStrcat()
  Appends a multibyte string to the destination string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.8 OCIMultiByteStrncmp()

Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods. Each string is in the specified length.

### Purpose

Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods. A length is specified for each string.

### Syntax

```
int OCIMultiByteStrncmp ( void           *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 these 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

The return values for the function are:

- 0, if `str1` = `str2`
- Positive, if `str1` > `str2`
- Negative, if `str1` < `str2`

**Related Topics**

- OCIMultiByteStrcpy()
  Copies a multibyte string into the destination array. It returns the number of bytes copied.
- OCIMultiByteStrncpy()
  Copies a specified number of bytes of a multibyte string into the destination array.
- OCIMultiByteStrcmp()
  Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.9 OCIMultiByteStrncpy()

Copies a specified number of bytes of a multibyte string into the destination array.

### Purpose

Copies a multibyte string into an array.

### Syntax

```
size_t OCIMultiByteStrncpy ( void            *hndl,
                             OraText         *dststr,
                             const OraText   *srcstr,
                             size_t           n );
```

### Parameters

**hndl (IN/OUT)**
Pointer to OCI environment or user session handle.

**dststr (IN)**
Pointer to the source multibyte string.

**srcstr (OUT)**
Pointer to the destination buffer.

**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 Topics

- OCIMultiByteStrcpy()
  Copies a multibyte string into the destination array. It returns the number of bytes copied.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.10 OCIMultiByteStrnDisplayLength()

Returns the number of display positions occupied by the multibyte string within the range of *n* bytes.

### Purpose

Returns the number of display positions occupied by the multibyte string within the range of `n` bytes.

### Syntax

```
size_t OCIMultiByteStrnDisplayLength ( void            *hndl,
                                       const OraText   *str1,
                                       size_t           n );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle.

**str1 (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.

### Related Topics

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.11 OCIMultiByteToWideChar()

Converts a `NULL`-terminated multibyte string into wide-character format.

### Purpose

Converts an entire `NULL`-terminated string into the wide-character string.

### Syntax

```
sword OCIMultiByteToWideChar ( void           *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 the 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 is `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 Topics**

- OCIWideCharToMultiByte()
  Converts a `NULL`-terminated wide-character string into a multibyte string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.12 OCIWideCharInSizeToMultiByte()

Converts part of a wide-character string to the multibyte string.

### Purpose

Converts part of a wide-character string to multibyte format.

### Syntax

```
sword OCIWideCharInSizeToMultiByte ( void            *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 needed 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`.

**Related Topics**

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.13 OCIWideCharMultiByteLength()

Determines the number of bytes required for a wide character in multibyte encoding.

**Purpose**

Determines the number of bytes required for a wide character in multibyte encoding.

**Syntax**

```
size_t OCIWideCharMultiByteLength ( void      *hndl,
                                    OCIWchar   wc );
```

**Parameters**

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` character.

**Comments**

If `OCI_UTF16ID` is specified for SQL `CHAR` data in function, then this function produces an error.

**Returns**

The number of bytes required for the wide character.

**Related Topics**

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.14 OCIWideCharStrCaseConversion()

Converts a wide-character string into the specified case and copies the result into the destination array.

**Purpose**

Converts a wide-character string into a specified case and copies the result into the destination array.

**Syntax**

```
size_t OCIWideCharStrCaseConversion ( void              *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

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`. 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 Topics

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.15 OCIWideCharStrcat()

Appends a wide-character string to the destination string.

### Purpose

Appends the wide-character string pointed to by `wsrcstr` to the wide-character string pointed to by `wdststr`.

### Syntax

```
size_t OCIWideCharStrcat ( void              *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 Topics**

- OCIWideCharStrncat()
  Appends, at most, *n* characters from a wide-character string to the destination string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.16 OCIWideCharStrchr()

Searches for the first occurrence of a wide character in a string. Returns a point to the wide character if the search is successful.

#### Purpose

Searches for the first occurrence of a specified character in a wide-character string.

#### Syntax

```
OCIWchar *OCIWideCharStrchr ( void             *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)**
The `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 Topics

- OCIWideCharStrrchr()
  Searches for the last occurrence of a character in a wide-character string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.17 OCIWideCharStrcmp()

Compares two wide-character strings by binary, linguistic, or case-insensitive comparison methods.

### Purpose

Compares two wide-character strings by binary (based on `wchar` encoding value), linguistic, or case-insensitive comparison methods.

### Syntax

```
int OCIWideCharStrcmp ( void              *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 these 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

The return values for the function are:

- 0, if `wstr1 = wstr2`
- Positive, if `wstr1 > wstr2`
- Negative, if `wstr1 < wstr2`

### Related Topics

- OCIWideCharStrncmp()
  Compares two wide-character strings by binary, linguistic, or case-insensitive methods. Each string is a specified length.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.18 OCIWideCharStrcpy()

Copies a wide-character string into a destination array. Returns the number of characters copied.

### Purpose

Copies a wide-character string into an array.

### Syntax

```
size_t OCIWideCharStrcpy ( void              *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 Topics**

- OCIWideCharStrncpy()
  Copies, at most, *n* characters of a wide-character string into the destination array.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.19 OCIWideCharStrlen()

Return the number of characters in a wide-character string.

### Purpose

Returns the number of characters in a wide-character string.

### Syntax

```
size_t OCIWideCharStrlen ( void             *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.

**Related Topics**

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.20 OCIWideCharStrncat()

Appends, at most, *n* characters from a wide-character string to the destination string.

### Purpose

Appends, at most, *n* characters from a wide-character string to the destination.

### Syntax

```
size_t OCIWideCharStrncat ( void             *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. The `wdststr` parameter 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 Topics

- OCIWideCharStrcat()
  Appends a wide-character string to the destination string.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.21 OCIWideCharStrncmp()

Compares two wide-character strings by binary, linguistic, or case-insensitive methods. Each string is a specified length.

#### Purpose

Compares two wide-character strings by binary, linguistic, or case-sensitive methods. Each string has a specified length.

#### Syntax

```
int OCIWideCharStrncmp ( void                *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 these 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

The return values for the function are:

- 0, if `wstr1 = wstr2`
- Positive, if `wstr1 > wstr2`
- Negative, if `wstr1 < wstr2`

**Related Topics**

- OCIWideCharStrcmp()
  Compares two wide-character strings by binary, linguistic, or case-insensitive comparison methods.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.22 OCIWideCharStrncpy()

Copies, at most, *n* characters of a wide-character string into the destination array.

### Purpose

Copies, at most, *n* characters from a wide-character string into a destination.

### Syntax

```
size_t OCIWideCharStrncpy ( void             *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 Topics

- OCIWideCharStrcpy()
  Copies a wide-character string into a destination array. Returns the number of characters copied.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.23 OCIWideCharStrrchr()

Searches for the last occurrence of a character in a wide-character string.

### Purpose

Searches for the last occurrence of a character in a wide-character string.

### Syntax

```
OCIWchar *OCIWideCharStrrchr ( void             *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)**
The `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

The `wchar` pointer if successful; otherwise, a `NULL` pointer.

#### Related Topics

- OCIWideCharStrchr()
  Searches for the first occurrence of a wide character in a string. Returns a point to the wide character if the search is successful.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

### 31.4.24 OCIWideCharToLower()

Converts a specified wide character into the corresponding lowercase character.

#### 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 ( void        *hndl,
                              OCIWchar     wc );
```

#### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `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 Topics

- OCIWideCharToUpper()
  Converts a specified wide character into the corresponding uppercase character.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.25 OCIWideCharToMultiByte()

Converts a `NULL`-terminated wide-character string into a multibyte string.

#### Purpose

Converts an entire `NULL`-terminated wide-character string into a multibyte string.

#### Syntax

```
sword OCIWideCharToMultiByte ( void             *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 the string.

**dst (OUT)**
Destination buffer for a multibyte string. The output buffer is `NULL`-terminated.

**src (IN)**
Source `wchar` string to be converted.

**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 Topics

- OCIMultiByteToWideChar()
  Converts a `NULL`-terminated multibyte string into wide-character format.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.4.26 OCIWideCharToUpper()

Converts a specified wide character into the corresponding uppercase character.

### 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 ( void      *hndl,
                              OCIWchar   wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `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 Topics

- OCIWideCharToLower()
  Converts a specified wide character into the corresponding lowercase character.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.5 OCI Character Classification Functions

Lists and describes OCI character classification functions.

Table 31-5 lists the OCI character classification functions that are described in this section.

**Table 31-5 OCI Character Classification Functions**

| Function | Purpose |
|---|---|
| OCIWideCharIsAlnum() | Test whether the wide character is a letter or a decimal digit |
| OCIWideCharIsAlpha() | Test whether the wide character is a letter |
| OCIWideCharIsCntrl() | Test whether the wide character is a control character |

**Table 31-5 (Cont.) OCI Character Classification Functions**

| Function | Purpose |
|---|---|
| OCIWideCharIsDigit() | Test whether the wide character is a decimal digital character |
| OCIWideCharIsGraph() | Test whether the wide character is a graph character |
| OCIWideCharIsLower() | Test whether the wide character is a lowercase character |
| OCIWideCharIsPrint() | Test whether the wide character is a printable character |
| OCIWideCharIsPunct() | Test whether the wide character is a punctuation character |
| OCIWideCharIsSingleByte() | Test whether the wide character is a single-byte character when converted to multibyte |
| OCIWideCharIsSpace() | Test whether the wide character is a space character |
| OCIWideCharIsUpper() | Test whether the wide character is an uppercase character |
| OCIWideCharIsXdigit() | Test whether the wide character is a hexadecimal digit |

## 31.5.1 OCIWideCharIsAlnum()

Tests whether the wide character is a letter or a decimal digit.

### Purpose

Tests whether a wide character is a letter or a decimal digit.

### Syntax

```
boolean OCIWideCharIsAlnum ( void       *hndl,
                             OCIWchar    wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.2 OCIWideCharIsAlpha()

Tests whether the wide character is a letter.

### Purpose

Tests whether a wide character is a letter.

### Syntax

```
boolean OCIWideCharIsAlpha ( void      *hndl,
                             OCIWchar   wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.3 OCIWideCharIsCntrl()

Tests whether the wide character is a control character.

### Purpose

Tests whether a wide character is a control character.

### Syntax

```
boolean OCIWideCharIsCntrl ( void        *hndl,
                             OCIWchar     wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.4 OCIWideCharIsDigit()

Test whether the wide character is a decimal digital character.

### Purpose

Tests whether a wide character is a decimal digit character.

### Syntax

```
boolean OCIWideCharIsDigit ( void       *hndl,
                             OCIWchar    wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.5 OCIWideCharIsGraph()

Tests whether a wide character is a graph character.

### Purpose

A graph character is a character with a visible representation and normally includes alphabetic letters, decimal digits, and punctuation.

### Syntax

```
boolean OCIWideCharIsGraph ( void      *hndl,
                             OCIWchar   wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.6 OCIWideCharIsLower()

Tests whether a wide character is a lowercase letter.

### Purpose

Tests whether a wide character is a lowercase letter.

### Syntax

```
boolean OCIWideCharIsLower ( void        *hndl,
                             OCIWchar    wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.7 OCIWideCharIsPrint()

Tests whether a wide character is a printable character.

### Purpose

Tests whether a wide character is a printable character.

### Syntax

```
boolean OCIWideCharIsPrint ( void       *hndl,
                             OCIWchar   wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.8 OCIWideCharIsPunct()

Tests whether a wide character is a punctuation character.

### Purpose

Tests whether a wide character is a punctuation character.

### Syntax

```
boolean OCIWideCharIsPunct ( void       *hndl,
                             OCIWchar    wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.9 OCIWideCharIsSingleByte()

Tests whether a wide character is a single-byte character when converted to multibyte.

### Purpose

Tests whether a wide character is a single-byte character when converted to multibyte.

### Syntax

```
boolean OCIWideCharIsSingleByte ( void      *hndl,
                                  OCIWchar   wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.10 OCIWideCharIsSpace()

Tests whether a wide character is a space character.

### Purpose

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 ( void     *hndl,
                             OCIWchar  wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.11 OCIWideCharIsUpper()

Tests whether a wide character is an uppercase letter.

### Purpose

Tests whether a wide character is an uppercase letter.

### Syntax

```
boolean OCIWideCharIsUpper ( void      *hndl,
                             OCIWchar   wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

## 31.5.12 OCIWideCharIsXdigit()

Tests whether the wide character is a hexadecimal digit.

### Purpose

Tests whether a wide character is a hexadecimal digit (0 through 9, A through F, a through f).

### Syntax

```
boolean OCIWideCharIsXdigit ( void      *hndl,
                              OCIWchar   wc );
```

### Parameters

**hndl (IN/OUT)**
OCI environment or user session handle to determine the character set.

**wc (IN)**
The `wchar` for testing.

### Returns

`TRUE` or `FALSE`.

# 31.6 OCI Character Set Conversion Functions

Lists and describes OCI character set conversion functions.

Conversion between Oracle Database 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 Database character set. Therefore, conversion back to the original character set is not always possible without data loss.

Table 31-6 lists the OCI character set conversion functions.

**Table 31-6 OCI Character Set Conversion Functions**

| Function | Purpose |
|---|---|
| OCICharSetConversionIsReplacementUsed() | Indicate whether replacement characters were used for characters that could not be converted in the last invocation of `OCINlsCharSetConvert()` or `OCICharSetToUnicode()` |
| OCICharSetToUnicode() | Convert a multibyte string to Unicode |
| OCINlsCharSetConvert() | Convert a string from one character set to another |
| OCIUnicodeToCharSet() | Convert a Unicode string into multibyte |

## 31.6.1 OCICharSetConversionIsReplacementUsed()

Indicates whether replacement characters were used for characters that could not be converted in the last invocation of `OCINlsCharSetConvert()` or `OCICharSetToUnicode()`.

### Purpose

Indicates whether the replacement character was used for characters that could not be converted during the last invocation of `OCICharSetToUnicode()` or `OCINlsCharSetConvert()`.

### Syntax

```
boolean OCICharSetConversionIsReplacementUsed ( void *hndl );
```

### Parameters

**hndl (IN/OUT)**
Pointer to an OCI environment or user session handle.

### Comments

Conversion between the Oracle Database 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 Database 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 `OCINlsCharSetConvert()` or `OCICharSetToUnicode()` was last invoked. Otherwise the function returns `FALSE`.

**Related Topics**

- OCINlsCharSetConvert()
  Converts a string from one character set to another.
- OCICharSetToUnicode()
  Converts a multibyte string to Unicode.

## 31.6.2 OCICharSetToUnicode()

Converts a multibyte string to Unicode.

### Purpose

Converts a multibyte string pointed to by `src` to Unicode out to the array pointed to by `dst`.

### Syntax

```
sword OCICharSetToUnicode ( void            *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 out to `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`.

### Related Topics

- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.6.3 OCINlsCharSetConvert()

Converts a string from one character set to another.

### 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 ( void          *hndl,
                             OCIError      *errhp,
                             ub2            dstid,
void           *dstp,
size_t          dstlen,
ub2             srcid,
const void     *srcp,
size_t          srclen,
size_t         *rsize );
```

### Parameters

**hndl (IN/OUT)**
Pointer to an OCI environment or user session handle.

**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` data type is necessary to hold strings in UTF-16.

### Returns

`OCI_SUCCESS` or `OCI_ERROR`; number of bytes converted.

### Related Topics

- OCINlsCharSetNameToId()
  Returns the Oracle Database character set ID for the specified Oracle Database character set name.

- OCICharSetConversionIsReplacementUsed()
  Indicates whether replacement characters were used for characters that could not be converted in the last invocation of `OCINlsCharSetConvert()` or `OCICharSetToUnicode()`.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 31.6.4 OCIUnicodeToCharSet()

Converts a Unicode string into multibyte string out to an array.

### Purpose

Converts a Unicode string to a multibyte string out to an array.

### Syntax

```
sword OCIUnicodeToCharSet ( void        *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 the 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 out to `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`.

**Related Topics**

- OCICharSetConversionIsReplacementUsed()
  Indicates whether replacement characters were used for characters that could not be converted in the last invocation of `OCINlsCharSetConvert()` or `OCICharSetToUnicode()`.
- OCIEnvNlsCreate()
  Creates and initializes an environment handle for OCI functions to work under.

## 31.7 OCI Messaging Functions

Lists and describes OCI messaging functions.

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

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide*

Table 31-7 lists the OCI messaging functions that are described in this section.

**Table 31-7 OCI Messaging Functions**

| Function | Purpose |
|---|---|
| OCIMessageClose() | Close a message handle and free any memory associated with the handle |
| OCIMessageGet() | Retrieve a message. If the buffer is not zero, then the function copies the message into the buffer |
| OCIMessageOpen() | Open a message handle in a specified language |

### 31.7.1 OCIMessageClose()

Closes a message handle and frees any memory associated with this handle.

**Purpose**

Closes a message handle and frees any memory associated with this handle.

**Syntax**

```
sword OCIMessageClose ( void      *hndl,
                        OCIError  *errhp,
                        OCIMsg    *msgh );
```

### Parameters

**hndl (IN/OUT)**
Pointer to an OCI environment or user session handle for the 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`.

### Related Topics

- OCIMessageOpen()
  Opens a message-handling facility in a specified language.
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 31.7.2 OCIMessageGet()

Gets a message with the given message number.

### 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 that 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.

### Related Topics

- OCIMessageOpen()
  Opens a message-handling facility in a specified language.

## 31.7.3 OCIMessageOpen()

Opens a message-handling facility in a specified language.

### Purpose

Opens a message-handling facility in a specified language.

### Syntax

```
sword OCIMessageOpen ( void             *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 the 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

`OCIMessageOpen()` 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`.

**Related Topics**

- OCIErrorGet()
  Returns an error message and an Oracle Database error code.

## 32

# OCI XML DB Functions

This chapter describes the OCI XML DB functions.

This chapter contains these topics:

- Introduction to XML DB Support in OCI
- OCI XML DB Functions

## 32.1 Introduction to XML DB Support in OCI

This chapter describes the XML DB functions in detail.

> **See Also:**
>
> OCI Support for XML

### 32.1.1 Conventions for OCI Functions

Describes conventions used in describing each function.

See the Conventions for OCI Functions for the conventions used in describing each function. The entries for each function may also contain the following information:

### 32.1.2 Returns

Describes the function return values.

Unless otherwise stated, the function returns the values described in Table 32-1.

**Table 32-1 Function Return Values**

| Return Value | Meaning |
|---|---|
| `OCI_SUCCESS` | The operation succeeded. |
| `OCI_ERROR` | The operation failed. The specific error can be retrieved by calling `OCIErrorGet()` on the error handle passed to the function. |
| `OCI_INVALID_HANDLE` | The OCI handle passed to the function is invalid. |

> ✎ **See Also:**
>
> Error Handling in OCI for more information about return codes and error handling

# 32.2 OCI XML DB Functions

Lists and describes OCI XML DB Functions.

Table 32-2 lists the OCI XML DB functions that are described in this chapter.

**Table 32-2 OCI XML DB Functions**

| Function | Purpose |
|---|---|
| OCIBinXmlCreateReposCtxFromConn() | Create metadata connection context |
| OCIBinXmlCreateReposCtxFromCPool() | Create metadata connection context connection pool |
| OCIBinXmlSetFormatPref() | Specify that images transferred are in binary XML format |
| OCIBinXmlSetReposCtxForConn() | Associate data connection with the metadata connection |
| OCIXmlDbFreeXmlCtx() | Free an XML context |
| OCIXmlDbInitXmlCtx() | Initialize an XML context for XML data from the database |

## 32.2.1 OCIBinXmlCreateReposCtxFromConn()

Creates a metadata connection context.

### Purpose

Creates a metadata connection context (`OCIBinXmlReposCtx`) from the specified `OCIEnv`/`OCISvcCtx` dedicated OCI connection. Note that this connection is dedicated to metadata use.

### Syntax

```
sword OCIBinXmlCreateReposCtxFromConn (   OCIEnv             *env,
                                          OCISvcCtx          *svcctx,
                                          OCIError           *err,
                                          OCIBinXmlReposCtx  **ctx);
```

### Parameters

**env (IN)**
The environment handle.

**svcctx (IN)**
The handle to the connection to be used to access the metadata.

**err (IN)**
The error handle.

**ctx (OUT)**
The metadata context that is created and returned.

### Returns

Returns -1 for error, 0 for success. The `err` parameter contains more information about the error.

### Related Topics

- OCIBinXmlCreateReposCtxFromCPool()
  Creates a metadata connection context from the specified connection pool.

## 32.2.2 OCIBinXmlCreateReposCtxFromCPool()

Creates a metadata connection context from the specified connection pool.

### Purpose

Creates a metadata connection context (`OCIBinXmlReposCtx`) from the specified connection pool. A connection from the connection pool is used whenever any information from the token repository is needed.

### Syntax

```
sword OCIBinXmlCreateReposCtxFromCPool (   OCIEnv              *env,
                                           OCICPool            *cpool,
                                           OCIError            *err,
                                           OCIBinXmlReposCtx  **ctx);
```

### Parameters

**env (IN)**
The environment handle.

**cpool (IN)**
The handle to the connection to be used to access the metadata.

**err (IN)**
The error handle.

**ctx (OUT)**
The metadata context that is created and returned.

### Returns

Returns -1 for error, 0 for success. The `err` parameter contains more information about the error.

### Related Topics

- OCIBinXmlCreateReposCtxFromConn()
  Creates a metadata connection context.

## 32.2.3 OCIBinXmlSetFormatPref()

Specifies that the images being transferred between client and server for the XML document be in binary XML format.

### Purpose

In the future, all communication will be in the binary XML format. Binary XML-aware applications can set this.

### Syntax

```
sword OCIBinXmlSetFormatPref (   xmldomdoc   *doc,
                                 ub4          formattype);
```

### Parameters

**doc (IN)**
The pointer to the `domdoc` to which the preference is to be applied.

**formattype (IN)**
The type of format to be used for pickling. Currently the only values allowed are `OCIXML_FORMATTYPE_TEXT` and `OCIXML_FORMATTYPE_BINXML`.

### Returns

Returns -1 for error, 0 for success.

**Related Topics**

- OCIBinXmlSetReposCtxForConn()
  Associates the data connection with the metadata connection.

## 32.2.4 OCIBinXmlSetReposCtxForConn()

Associates the data connection with the metadata connection.

### Purpose

Note that with a dedicated connection, the environment handle must be the same for the data connection and for the metadata connection.

### Syntax

```
sword OCIBinXmlSetReposCtxForConn (   OCISvcCtx          *dataconn,
                                      OCIBinXmlReposCtx  *reposctx);
```

### Parameters

**dataconn (IN)**
The data connection handle.

**reposctx (IN)**
The pointer to the metadata connection.

#### Returns

Returns -1 for error, 0 for success. The `err` parameter contains more information about the error.

#### Related Topics

- OCIBinXmlSetFormatPref()
  Specifies that the images being transferred between client and server for the XML document be in binary XML format.

## 32.2.5 OCIXmlDbFreeXmlCtx()

Frees an XML context.

#### Purpose

Frees any allocations made by the call to `OCIXmlDbInitXmlCtx()`.

#### Syntax

```
void OCIXmlDbFreeXmlCtx ( xmlct   *xctx );
```

#### Parameters

**xctx (IN)**
The XML context to terminate.

#### Comments

> **See Also:**
>
> Using OCI XML DB Functions for a usage example

#### Returns

Returns -1 for error, 0 for success.

#### Related Topics

- OCIXmlDbInitXmlCtx()
  Initializes an XML context.

## 32.2.6 OCIXmlDbInitXmlCtx()

Initializes an XML context.

#### Purpose

Initializes 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 either the OCI duration, in which the default value is `OCI_DURATION_SESSION`, or to an error handler that 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 numeric 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:**
>
> Using OCI XML DB Functions 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 Topics

- OCIXmlDbFreeXmlCtx()
  Frees an XML context.

# 33

# Oracle ODBC Driver

The Oracle ODBC Driver enables ODBC applications on Microsoft Windows, as well as UNIX platforms like Linux, Solaris, IBM Advanced Interactive eXecutive (AIX), HP-UX Itaniutm, and IBM Linux on Platform z read and write access to Oracle® databases through the ODBC interface using Oracle Net Services software.

This chapter is a placeholder to indicate that information about using the Oracle ODBC Driver can be found in *Oracle Database Development Guide* and in the Oracle ODBC Help set.

# 34

# Introduction to the OCI Interface for XStream

This chapter provides an overview of the OCI interface for XStream.

The Oracle Call Interface (OCI) includes an interface for XStream. This chapter provides an introduction to the OCI interface for XStream.

This chapter contains these topics:

- About the XStream Interface
- Handler and Descriptor Attributes

> **See Also:**
>
> - *Oracle Database XStream Guide* for detailed information about XStream Out concepts
> - *Oracle Database XStream Guide* for information about configuring XStream Out
> - OCI XStream Functions for information about OCI XStream functions

## 34.1 About the XStream Interface

Since Oracle Database 11g Release 2, APIs, known as XStream Out and XStream In, are available.

This technology enables high performance, near real-time information-sharing infrastructure between Oracle databases and non-Oracle databases, non-RDBMS Oracle products, file systems, third party software applications, and so on. XStream is built on the infrastructure used by Oracle Streams.

> **See Also:**
>
> OCI XStream functions in OCI XStream Functions

### 34.1.1 XStream Out

XStream Out allows a remote client to attach to an outbound server and extract row changes in the form of logical change records (LCRs).

To use XStream Out, a capture process and an outbound server must be created. All data types supported by Oracle Streams, including LOB, `LONG`, and `XMLType`, are supported by XStream. The capture process and the outbound server need not be on the same database instance. After the capture process and the outbound server have started, row changes are captured and sent to the outbound server. An external client application can connect to this outbound server using OCI. After the connection is established, the client application can loop while waiting for LCRs from the outbound server. The client application can register a client-side callback to be invoked each time an LCR is received. At any time, the client application can detach from the outbound server as needed. Upon restart, the outbound server knows where in the redo stream to start streaming LCRs to the client application.

> **See Also:**
>
> - *Oracle Database XStream Guide* for an introduction to XStream Out

## 34.1.2 XStream In

To replicate non-Oracle data into Oracle databases, use XStream In.

This technology allows a remote client application to attach to an inbound server and send row and DDL changes in the form of LCRs.

An external client application connects to the inbound server using OCI. After the connection is established, the client application acts as the capture agent for the inbound server by streaming LCRs to it. A client application can attach to only one inbound server for each database connection, and each inbound server only allows one client application to attach to it.

> **See Also:**
>
> XStream In concepts in *Oracle Database XStream Guide*

## 34.1.3 Position Order and LCR Streams

Each LCR has a position attribute.

The position of an LCR identifies its placement in the stream of LCRs in a transaction.

> **See Also:**
>
> Position order in an LCR in *Oracle Database XStream Guide*

## 34.1.4 XStream and Character Sets

For XStream Out, in general, setting the client application character set to the outbound server database character set is the best practice.

XStream Out implicitly converts character data in LCRs from the outbound server database character set to the client application character set. XStream In implicitly converts character data in LCRs from the client application character set to the inbound server database character set.

To improve performance, complete the following tasks:

- Analyze the LCR data flow from the source to the destination.
- Set the client character set of the OCI client application to the one that minimizes character conversion, incurs no data loss, and takes advantage of the implicit conversion done by XStream or the destination.

# 34.2 Handler and Descriptor Attributes

Describes the attributes for OCI handles and descriptors.

The attributes for OCI handles and descriptors can be read with `OCIAttrGet()` and modified with `OCIAttrSet()`.

## 34.2.1 Conventions

For each handle type, the attributes that 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 Data Type**
This is the data type of the attribute. If necessary, a distinction is made between the data type for `READ` and `WRITE` modes.

## 34.2.2 Server Handle Attributes

Lists the server handle attributes.

The following server handle attributes are available:

- OCI_ATTR_XSTREAM_ACK_INTERVAL
- OCI_ATTR_XSTREAM_IDLE_TIMEOUT

### 34.2.2.1 OCI_ATTR_XSTREAM_ACK_INTERVAL

Describes the OCI_ATTR_XSTREAM_ACK_INTERVAL server handle attribute.

**Mode**
`READ/WRITE`

**Description**
For XStream Out, the ACK interval is the minimum interval in seconds that the outbound server receives the processed low position from the client application. After each ACK interval, the outbound server ends any in-progress `OCIXStreamOutLCRReceive()` or `OCIXStreamOutLCRCallbackReceive()` call so that the processed low position cached at the client application can be sent to the outbound server.
For XStream In, the ACK interval is the minimum interval in seconds that the inbound server sends the processed low position to the client application. After each ACK interval, any in-progress `OCIXStreamInLCRSend()` or `OCIXStreamInLCRCallbackSend()` call is terminated for the inbound server to send a new processed low position to the client application.
The default value for `OCI_ATTR_XSTREAM_ACK_INTERVAL` is 30 seconds. This attribute is checked only during the `OCIXStreamOutAttach()` or `OCIXStreamInAttach()` calls. Thus, it must be set before invoking these APIs; otherwise, the default value is used.

**Attribute Data Type**
`ub4 */ub4`

### 34.2.2.2 OCI_ATTR_XSTREAM_IDLE_TIMEOUT

Describes the OCI_ATTR_XSTREAM_IDLE_TIMEOUT server handle attribute.

**Mode**
`READ/WRITE`

**Description**
The idle timeout is the number of seconds of idle the outbound server waits for an LCR before terminating the `OCIXStreamOutLCRReceive()` or `OCIXStreamOutLCRCallbackReceive()` call.
The default for `OCI_ATTR_XSTREAM_IDLE_TIMEOUT` is one second. This attribute is checked only during the `OCIXStreamOutAttach()` or `OCIXStreamInAttach()` call. Thus, it must be set before invoking these APIs; otherwise, the default value is used.

**Attribute Data Type**
`ub4 */ub4`

# 35

# OCI XStream Functions

This chapter describes the XStream functions for OCI.

A row logical change record (LCR) is used to encapsulate each row change. It includes the schema name, table name, DML operation, and the column values. For update operations, both before and after column values are included. The column data is in the format specified by the "Program Variable" column in Table 35-3. Character columns are converted to the client's character set.

A DDL LCR is used to encapsulate each DDL change. It includes the object name, the DDL text, and the DDL command, for example, `ALTER TABLE` or `TRUNCATE TABLE`. See Table A-1 for a list of DDL command codes.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information about NLS settings.
>
> XStream sample programs are found in `xstream/oci` under the `$ORACLE_HOME/demo` directory.

Each LCR also has a transaction ID and position. For transactions captured outside Oracle databases, any byte-comparable `RAW` array can be used as the LCR position, if the position of each LCR in the stream is strictly increasing.

This chapter contains the topic:

- About Using the XStream Interface
- Introduction to XStream Functions
- OCI XStream Functions

## 35.1 About Using the XStream Interface

Since Oracle Database 11g Release 2, Oracle Streams provides enhanced APIs, known as eXtended Streams (XStream) Out and XStream In, to enable high performance, near real-time information-sharing infrastructure between Oracle databases and non-Oracle databases, non-RDBMS Oracle products, file systems, third party software applications, and so on.

XStream is built on top of Streams infrastructure.

This section includes the following topics:

- XStream Out
- XStream In
- Security of XStreams

**See Also:**

OCI XStream Functions

## 35.1.1 XStream Out

XStream Out allows a remote client to attach to an outbound server (a Streams apply process) and extract row changes in the form of Logical Change Records (LCRs).

To use XStream Out, a capture and an apply process must be created similar to other Streams setup. All data types supported by Oracle Streams including LOB, `LONG`, and `XMLType` are supported by XStreams. Such an apply process is called an outbound server. The capture and the outbound server may or may not be on the same database instance. After the capture and the outbound server have started, row changes will be captured and sent to the outbound server. An external client can then connect to this outbound server using OCI. After the connection is established, the client can loop waiting for LCRs from the outbound server. The client can register a client-side callback to be invoked each time an LCR is received. At anytime, the client can detach from the outbound server as needed. Upon restart, the outbound server knows where in the redo stream to start streaming LCRs to the client.

This section includes the following topics:

- LCR Streams
- The Processed Low Position and Restart Considerations

**See Also:**

- *Oracle Database XStream Guide* for more details of XStreams concepts

### 35.1.1.1 LCR Streams

Describes characteristics of an LCR stream.

- An LCR stream must be repeatable.
- An LCR stream must contain a list of assembled and committed transactions.
- LCRs from one transaction are contiguous. There is no interleaving of transactions in the LCR stream.
- Each transaction within an LCR stream must have an ordered list of LCRs and a transaction ID.
- The last LCR in each transaction must be a commit LCR.
- Each LCR must have a unique position.
- The position of all LCRs within a single transaction and across transactions must be strictly increasing.

### 35.1.1.2 The Processed Low Position and Restart Considerations

If the outbound server or the client aborts abnormally, the connection between the two is automatically broken. The client needs to maintain the processed low position to properly recover after a restart.

The processed low position is a position below which all LCRs have been processed by the client. This position should be maintained by the client while applying each transaction. Periodically this position is sent to the server while the client executes XStream Out APIs. This position indicates to the server that the client has processed all LCRs below or equal to this position; thus, the server can purge redo logs that are no longer needed.

Upon restart, the client must re-attach to the outbound server. During the attach call, the client can notify the outbound server of the last position received by the client. The outbound server then sends LCRs with position greater than this last position. If the client does not specify the last position (that is, a `NULL` is specified), the outbound server will retrieve the processed low position from its system tables and derive the starting position to mine the redo logs. It will send to the client the LCRs with position greater than this processed low position.

## 35.1.2 XStream In

To replicate non-Oracle data into Oracle databases use XStream In which allows a remote client to attach to an inbound server (a Streams apply process) and send row and DDL changes in the form of LCRs.

An external client application connects to the inbound server using OCI. After the connection is established, the client application acts as the capture agent for the inbound server by streaming LCRs to it. A client application can attach to only one inbound server per database connection. Each inbound server only allows one client attaching to it.

XStream In uses the following features of Oracle Streams:

- High performance processing of DML changes using an apply process and, optionally, apply process parallelism.
- Apply process features such as SQL generation, conflict detection and resolution, error handling, and customized processing with apply handlers.
- Streaming network transmission of information with minimal network round trips.

XStream In supports all of the data types that are supported by Oracle Streams, including LOBs, `LONG`, `LONG RAW`, and `XMLType`. A client application sends LOB and `XMLType` data to the inbound server in chunks. Several chunks make up a single column value of LOB or `XMLType`.

This section includes the following topics:

- Processed Low Position and Restart Ability
- Stream Position

### 35.1.2.1 Processed Low Position and Restart Ability

The processed low position is the position below which the inbound server no longer requires any LCRs.

This position corresponds with the oldest SCN for an Oracle Streams apply process that applies changes captured by a capture process.

The processed low position indicates that the LCRs less than or equal to this position have been processed by the inbound server. If the client re-attaches to the inbound server, it only needs to send LCRs greater than the processed low position because the inbound server discards any LCRs that are less than or equal to the processed low position.

If the client application aborts abnormally, then the connection between the client application and the inbound server is automatically broken. Upon restart, the client application retrieves the processed low position from the inbound server and instructs its capture agent to retrieve changes starting from this processed low position.

To limit the recovery time of a client application using the XStream In interface, the client application can send activity, such as empty transactions, periodically to the inbound server. When there are no LCRs to send to the server, the client can send a row LCR with a commit directive to advance the inbound server's processed low position. This activity acts as an acknowledgement so that the inbound server's processed low position can be advanced. The LCR stream sent to an inbound server must follow the LCR stream properties for XStream Out defined above.

### 35.1.2.2 Stream Position

Stream position refers to the position of an LCR in a given LCR stream.

For transactions captured outside Oracle databases the stream position must be encoded in certain format (for example, base-16 encoding) that supports byte comparison. The stream position is key to the total order of transaction stream sent by clients using the XStream In interface.

## 35.1.3 Security of XStreams

Describes the security characteristics of XStreams.

XStream Out allows regular users to receive LCRs without requiring system level privileges. System level privileges, such as DBA role, are required to configure XStream Out. The user who configures XStream Out can specify a regular user as the connect user who can attach to an outbound server to receive LCRs.

XStream In allows regular users to update tables in its own schema without requiring system level privileges (for example, DBA) to configure XStream In.

XStream cannot assume that the connected user to the inbound or outbound server is trusted.

OCI clients must connect to an Oracle database prior to attaching to an XStream outbound or inbound server created on that database. The connected user must be the same as the `connect_user` configured for the attached outbound server or the `apply_user` configured for the attached inbound server; otherwise, an error is raised.

**See Also:**

*Oracle Database XStream Guide* for more about configuring Oracle XStreams

## 35.2 Introduction to XStream Functions

Lists and describes the conventions used to describe the functions.

These conventions include:

- 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 in the following table.

**Table 35-1 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), which can include return values, restrictions on the use of the function, examples, or other information that can be useful when using the function in an application.

## 35.3 OCI XStream Functions

Lists and describes OCI XStream Functions.

This section and Table 35-2 describe the OCI XStream functions.

**Table 35-2 OCI XStream Functions**

| Function | Purpose |
|---|---|
| **LCR Functions** | To get and set one or more values of an LCR. **Note:** These calls do not require a server round-trip. |
| OCILCRAttributesGet() | Gets extra attribute information in (ROW or DDL) LCR. |
| OCILCRAttributesSet() | Populates extra attribute information in ROW or DDL LCR. |

**Table 35-2 (Cont.) OCI XStream Functions**

| Function | Purpose |
|---|---|
| OCILCRComparePosition() | Compares two position LCRID values. |
| OCILCRConvertPosition() | Converts an LCRID value to the specified version (1 or 2). |
| OCILCRFree() | Frees the LCR |
| OCILCRHeaderGet() | Returns the common header fields for a ROW or DDL LCR |
| OCILCRHeaderSet() | Initializes the common header fields for a ROW or DDL LCR |
| OCILCRDDLInfoGet() | Retrieves specific fields in a DDL LCR |
| OCILCRDDLInfoSet() | Populates DDL-specific fields in a DDL LCR |
| OCILCRLobInfoGet() | Returns the LOB information for a piece-wise LOB LCR |
| OCILCRLobInfoSet() | Sets the LOB information for a piece-wise LOB LCR |
| OCILCRNew() | Constructs a new LCR object of the specified type (row or DDL) for the given duration |
| OCILCRRowColumnInfoGet() | Returns the column fields in a row LCR |
| OCILCRRowColumnInfoSet() | Populates column fields in a row LCR |
| OCILCRRowStmtGet() | Returns the generated SQL statement for the row LCR, with values in-lined |
| OCILCRRowStmtWithBindVarGet() | Returns the generated SQL statement, which uses bind variables for column values |
| OCILCRSCNsFromPosition() | Gets the SCN and commit SCN from a position value |
| OCILCRSCNToPosition() | Converts SCN to position |
| OCILCRWhereClauseGet() | Gets the `WHERE` clause statement for the given row LCR |
| OCILCRWhereClauseWithBindVarGet() | Gets the `WHERE` clause statement with bind variables for the given row LCR |
| **XStream In Functions** | To send an LCR stream to an XStream inbound server |
| OCIXStreamInAttach() | Attaches to an inbound server |
| OCIXStreamInChunkSend() | Sends chunk data to the inbound server |
| OCIXStreamInCommit() | Commits the given transaction |
| OCIXStreamInDetach() | Detaches from the inbound server |
| OCIXStreamInErrorGet() | Returns the first error encountered by the inbound server since the attach call |
| OCIXStreamInFlush() | Flushes the network while attaching to an XStream inbound server |

**Table 35-2 (Cont.) OCI XStream Functions**

| Function | Purpose |
|---|---|
| OCIXStreamInLCRCallbackSend() | Sends the LCR stream to the attached inbound server using callbacks |
| OCIXStreamInLCRSend() | Sends the LCR stream to the attached inbound server using callbacks |
| OCIXStreamInProcessedLWMGet() | Gets the local processed low position |
| OCIXStreamInSessionSet() | Sets session attributes for XStream In functions |
| **XStream Out Functions** | To receive an LCR stream from an XStream outbound server |
| OCIXStreamOutAttach() | Attaches to an outbound server |
| OCIXStreamOutChunkReceive() | Retrieves data of each LOB or `LONG` or `XMLType` column one chunk at a time |
| OCIXStreamOutDetach() | Detaches from the outbound server |
| OCIXStreamOutLCRCallbackReceive() | Gets the LCR stream from the outbound server using callbacks |
| OCIXStreamOutLCRReceive() | Receives an LCR stream from an outbound server without using callbacks |
| OCIXStreamOutProcessedLWMSet() | Updates the local copy of the processed low-water mark |
| OCIXStreamOutSessionSet() | Sets session attributes for XStream Out functions |

## 35.3.1 OCILCRAttributesGet()

Gets extra attribute information in (ROW or DDL) LCR.

### Purpose

It gets any extra non-first class attributes that are not populated through `OCILCRHeaderGet()`, `OCILCRDDLInfoGet()`, or `OCILCRProcedureInfoGet()`, for example, edition name.

### Syntax

```
sword OCILCRAttributesGet (      OCISvcCtx   *svchp,
                                 OCIError    *errhp,
                                 ub2         *num_attrs,
                                 oratext     **attr_names,
                                 ub2         *attr_namesl,
                                 ub2         *attr_dtyp,
                                 void        **attr_valuesp,
                                 OCIInd      *attr_indp,
                                 ub2         *attr_alensp,
                                 void        *lcrp,
                                 ub2          array_size,
                                 ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**num_attrs (OUT)**
Number of extra attributes.

**attr_names (OUT)**
An array of extra attribute name pointers.

**attr_namesl (OUT)**
An array of extra attribute name lengths.

**attr_dtyp (OUT)**
An array of extra attribute data types. Valid data types: see Comments.

**attr_valuesp (OUT)**
An array of extra attribute data value pointers.

**attr_indp (OUT)**
An indicator array. Each returned element is an `OCIInd` value (`OCI_IND_NULL` or `OCI_IND_NOTNULL`).

**attr_alensp (OUT)**
An array of actual extra attribute data lengths. Each element in `alensp` is the length in bytes.

**lcrp (IN)**
Pointer to ROW or DDL LCR.

**array_size (IN)**
Size of the array argument in the other parameters. If `array_size` is not large enough to accommodate the number of attributes in the requested attribute list, then `OCI_ERROR` is returned. Parameter `num_attrs` returns the expected size.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

The valid data types for `attr_dtyp` are:

```
SQLT_CHR
SQLT_INT
SQLT_RDD
```

## 35.3.2 OCILCRAttributesSet()

Populates extra attribute information in ROW or DDL LCR.

### Purpose

It populates any extra non-first class attributes that cannot be set through `OCILCRHeaderSet()`, `OCILCRDDLInfoSet()`, or `OCILCRRowColumnInfoSet()`, for example, edition name.

### Syntax

```
sword OCILCRAttributesSet (      OCISvcCtx   *svchp,
                                 OCIError    *errhp,
                                 ub2          num_attrs,
                                 oratext    **attr_names,
                                 ub2         *attr_names_lens,
                                 ub2         *attr_dtyp,
                                 void       **attr_valuesp,
                                 OCIInd      *attr_indp,
                                 ub2         *attr_alensp,
                                 void        *lcrp,
                                 ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**num_attrs (IN)**
Number of extra attributes.

**attr_names (IN)**
Pointer to an array of extra attribute names. Attribute names must be canonicalized.

**attr_names_lens (IN)**
Pointer to an array of extra attribute name lengths.

**attr_dtyp (IN)**
Pointer to an array of extra attribute data types. See valid data types in Comments of `OCILCRRowColumnInfoSet()`.

**attr_valuesp (IN)**
Address of an array of extra attribute data values.

**attr_indp (IN)**
Pointer to an indicator array. For all data types, this is a pointer to an array of `OCIInd` values (`OCI_IND_NULL` or `OCI_IND_NOTNULL`).

**attr_alensp (IN)**
Pointer to an array of actual extra attribute data lengths. Each element in `attr_lensp` is the length in bytes.

**lcrp (IN/OUT)**
Pointer to a ROW or DDL LCR.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

Valid attributes are:

```
#define OCI_LCR_ATTR_THREAD_NO              "THREAD#"
#define OCI_LCR_ATTR_ROW_ID                 "ROW_ID"
#define OCI_LCR_ATTR_SESSION_NO             "SESSION#"
#define OCI_LCR_ATTR_SERIAL_NO              "SERIAL#"
#define OCI_LCR_ATTR_USERNAME               "USERNAME"
#define OCI_LCR_ATTR_TX_NAME                "TX_NAME"
#define OCI_LCR_ATTR_EDITION_NAME           "EDITION_NAME"
#define OCI_LCR_ATTR_MESSAGE_TRACKING_LABEL "MESSAGE_TRACKING_LABEL"
#define OCI_LCR_ATTR_CURRENT_USER           "CURRENT_USER"
#define OCI_LCR_ATTR_ROOT_NAME              "ROOT_NAME"
```

### Related Topics

- OCILCRRowColumnInfoSet()
  Populates column fields in a row LCR.

## 35.3.3 OCILCRComparePosition()

Compares two position LCRID values.

### Purpose

These LCRIDs can have different versions. The provided position must be a valid LCRID for Oracle Database 12*c* Release 2 (12.2).

### Syntax

```
sword OCILCRComparePosition(OCISvcCtx  *svchp,
                            OCIError   *errhp,
                            ub1        *position1,
                            ub2         position1_len,
                            ub1        *position2,
                            ub2         position2_len,
                            ub4         mode,
                            sb2        *result);
```

### Parameters

**svchp (IN)**
OCI service context.

**errhp (IN)**
OCI Error handle.

**position1 (IN)**
The first position value to compare.

**position1_len (IN)**
The length of `position1`.

**position2 (IN)**
The second position value to compare.

**position2_len (IN)**
The length of `position2`.

**mode (IN)**
The mode flags.

**result (OUT)**
`0` if both values are equal.
`-1` if `position1` is less than `position2`.
`1` if `position1` is greater than `position2`.

### Returns

`OCI_SUCCESS` if the conversion succeeds, `OCI_ERROR` otherwise.

### Usage Notes

Supported modes are:

- `0` - Complete byte comparison.
- `2` - Smaller length is smaller value

## 35.3.4 OCILCRConvertPosition()

Converts an LCRID value to the specified version (1 or 2).

### Purpose

The provided LCRID must be valid for Oracle Database 12*c* Release 2 (12.2).

### Syntax

```
sword OCILCRConvertPosition(OCISvcCtx  *svchp,
                            OCIError   *errhp,
                            ub1        *in_position,
                            ub2         in_position_len,
                            ub1        *out_position,
                            ub2        *out_position_len,
                            ub1         to_version,
                            ub4         mode);
```

### Parameters

**svchp (IN)**
OCI service context.

**errhp (IN)**
OCI Error handle.

**in_position (IN)**
The position value to convert.

**in_position_len (IN)**
The length of `in_position`.

**out_position (OUT)**
The result position value (in the specified version). The memory must be preallocated.

**out_position_len (OUT)**
The length of `out_position`.

**to_version (IN)**
The version to which to convert to. The value `1` should be specified for the original `OCI_LCRID_V1` format. The value `2` should be specified for the `OCI_LCRID_V2` format.

```
#define  OCI_LCRID_V1       1
#define  OCI_LCRID_V2       2
```

**mode (IN)**
The mode flags.

#### Returns

`OCI_SUCCESS` if the conversion succeeds.

`OCI_SUCCESS` if the LCRID is already in the desired version.

`OCI_ERROR` if the conversion fails.

#### Usage Notes

None.

## 35.3.5 OCILCRFree()

Frees the LCR.

#### Purpose

Frees the LCR.

#### Syntax

```
sword OCILCRFree ( OCISvcCtx   *svchp,
                   OCIError    *errhp,
                   void        *lcrp,
                   ub4          mode );
```

#### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**lcrp (IN/OUT)**
Streams LCR pointer.

**mode (IN)**
Specify `OCI_DEFAULT`.

## 35.3.6 OCILCRDDLInfoGet()

Retrieves specific fields in a DDL LCR.

### Purpose

Retrieves specific fields in a DDL LCR.

### Syntax

```
sword OCILCRDDLInfoGet ( OCISvcCtx    *svchp,
                         OCIError     *errhp,
                         oratext     **object_type,
                         ub2          *object_type_len,
                         oratext     **ddl_text,
                         ub4          *ddl_text_len,
                         oratext     **logon_user,
                         ub2          *logon_user_len,
                         oratext     **current_schema,
                         ub2          *current_schema_len,
                         oratext     **base_table_owner,
                         ub2          *base_table_owner_len,
                         oratext     **base_table_name,
                         ub2          *base_table_name_len,
                         oraub8       *flag,
                         void         *ddl_lcrp,
                         ub4           mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**object_type (OUT)**
The type of object on which the DDL statement was executed. (See OCILCRDDLInfoSet().) Optional. If not `NULL`, then both `object_type` and `object_type_len` arguments must not be `NULL`.

**object_type_len (OUT)**
Length of the `object_type` string without the `NULL` terminator.

**ddl_text (OUT)**
The text of the DDL statement. Optional. If not `NULL`, then both `ddl_text` and `ddl_text_len` arguments must not be `NULL`.

**ddl_text_len (OUT)**
DDL text length in bytes without the `NULL` terminator.

**logon_user (OUT)**
Canonicalized (follows a rule or procedure) name of the user whose session executed the DDL statement. Optional. If not `NULL`, then both `logon_user` and `logon_user_len` arguments must not be `NULL`.

**logon_user_len (OUT)**
Length of the logon_user string without the `NULL` terminator.

**current_schema (OUT)**
The canonicalized schema name that is used if no schema is specified explicitly for the modified database objects in `ddl_text`. Optional. If not `NULL`, then both `current_schema` and `current_schema_len` arguments must not be `NULL`.

**current_schema_len (OUT)**
Length of the `current_schema` string without the `NULL` terminator.

**base_table_owner (OUT)**
If the DDL statement is a table-related DDL (such as `CREATE TABLE` and `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_owner` specifies the canonicalized owner of the table involved. Otherwise, `base_table_owner` is `NULL`. Optional. If not `NULL`, then both `base_table_owner` and `base_table_owner_len` arguments must not be `NULL`.

**base_table_owner_len (OUT)**
Length of the `base_table_owner` string without the `NULL` terminator.

**base_table_name (OUT)**
If the DDL statement is a table-related DDL (such as `CREATE TABLE` and `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_name` specifies the canonicalized name of the table involved. Otherwise, `base_table_name` is `NULL`. Optional. If not `NULL`, then both `base_table_name` and `base_table_name_len` arguments must not be `NULL`.

**base_table_name_len (OUT)**
Length of the `base_table_name` string without the `NULL` terminator.

**flag (OUT)**
DDL LCR flag. Optional. Data not returned if argument is `NULL`. Future extension not used currently.

**ddl_lcrp (IN)**
DDL LCR. Cannot be `NULL`.

**mode (IN)**
Specify `OCI_DEFAULT`.

**Related Topics**

- OCILCRDDLInfoSet()
  Populates DDL-specific fields in a DDL LCR.

## 35.3.7 OCILCRHeaderGet()

Returns the common header fields for ROW or DDL LCR.

### Purpose

All returned pointers point directly to the corresponding LCR fields.

### Syntax

```
sword OCILCRHeaderGet ( OCISvcCtx    *svchp,
                        OCIError     *errhp,
                        oratext     **src_db_name,
                        ub2          *src_db_name_len,
                        oratext     **cmd_type,
                        ub2          *cmd_type_len,
                        oratext     **owner,
                        ub2          *owner_len,
                        oratext     **oname,
                        ub2          *oname_len,
                        ub1         **tag,
                        ub2          *tag_len,
                        oratext     **txid,
                        ub2          *txid_len,
                        OCIDate      *src_time,
                        ub2          *old_columns,
                        ub2          *new_columns,
                        ub1         **position,
                        ub2          *position_len,
                        oraub8       *flag,
                        void         *lcrp,
                        ub4           mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**src_db_name (OUT)**
Canonicalized source database name. Must be non-`NULL`.

**src_db_name_len (OUT)**
Length of the `src_db_name` string in bytes excluding the `NULL` terminator.

**cmd_type (OUT)**
For row LCRs: One of the following values:

> **Note:**
>
> The values, `#define OCI_LCR_ROW_CMD_ROLLBACK` and `#define OCI_LCR_ROW_CMD_START_TX`, is functionality that is available starting with Oracle Database 11*g* Release 2 (11.2.0.2).

```
#define OCI_LCR_ROW_CMD_INSERT
#define OCI_LCR_ROW_CMD_DELETE
#define OCI_LCR_ROW_CMD_UPDATE
#define OCI_LCR_ROW_CMD_COMMIT
#define OCI_LCR_ROW_CMD_ROLLBACK
#define OCI_LCR_ROW_CMD_START_TX
#define OCI_LCR_ROW_CMD_LOB_WRITE
#define OCI_LCR_ROW_CMD_LOB_TRIM
#define OCI_LCR_ROW_CMD_LOB_ERASE
```

For DDL LCRs: One of the command types in OCI_ATTR_SQLFNCODE.

**cmd_type_len (OUT)**
Length of the `cmd_type` string in bytes excluding the `NULL` terminator.

**owner (OUT)**
Canonicalized table owner name. Must be non-`NULL`. For procedure LCRs, the package or procedure owner is returned in `owner`.

**owner_len (OUT)**
Length of the `owner` string in bytes excluding the `NULL` terminator.

**oname (OUT)**
Canonicalized table name. Must be non-`NULL`. For procedure LCRs, the procedure name is returned in `oname`.

**oname_len (OUT)**
Length of the `oname` string in bytes excluding the `NULL` terminator.

**tag (OUT)**
A binary tag that enables tracking of the LCR. For example, you can use this tag to determine the original source database of the DML statement if apply forwarding is used.

**tag_len (OUT)**
Number of bytes in the tag.

**txid (OUT)**
Transaction ID. Must be non-`NULL`

**txid_len (OUT)**
Length of the string in bytes excluding the `NULL` terminator.

**src_time (OUT)**
The time when the change was generated in the redo log file of the source database.

**old_columns (OUT)**
Number of columns in the `OLD` column list. Returns 0 if the input LCR is a DDL LCR. Optional.

**new_columns (OUT)**
Number of columns in the `NEW` column list. Returns 0 if the input LCR is a DDL LCR. Optional.

**position (OUT)**
Position for LCR.

**position_len (OUT)**
Length of `position`.

**flag (OUT)**
LCR flag. Possible flags are listed in Comments.

**lcrp (IN)**
`lcrp` cannot be `NULL`.

**mode (IN)**
`OCILCR_NEW_ONLY_MODE` - If this mode is specified, then the `new_columns` returned is the count of the columns in the `NEW` column list only. Otherwise, the `new_columns` returned is the number of distinct columns present in either the `NEW` or the `OLD` column list of the given row LCR.
`OCI_LCR_APPCON_REPLAY` - If this mode is specified, then it indicates the LCR is replayed from the application container sync statement.

### Comments

LCR flag.

```
#define OCI_ROWLCR_HAS_ID_KEY_ONLY  /* only has ID key cols */
#define OCI_ROWLCR_SEQ_LCR          /* sequence lcr */
```

`OCILCRProcedureInfoGet()` can be called to get the package name.

## 35.3.8 OCILCRRowStmtGet()

Returns the generated SQL statement for the row LCR, with values in-lined.

### Purpose

Users must preallocate the memory for `sql_stmt`, and `*sql_stmt_len` must be set to the size of the allocated buffer, when it is passed in. If `*sql_stmt_len` is not large enough to hold the generated SQL statement, then an error is raised.

### Syntax

```
sword OCILCRRowStmtGet ( OCISvcCtx   *svchp,
                         OCIError    *errhp,
                         oratext     *row_stmt,
                         ub4         *row_stmt_len,
                         void        *row_lcrp,
                         ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**row_stmt (IN/OUT)**
The generated SQL statement for the row LCR.

**row_stmt_len (IN/OUT)**
Set to the size of the allocated buffer for `row_stmt` when passed in; returns the length of `row_stmt`.

**row_lcrp (IN)**
Pointer to row LCR.

**mode (IN)**
Specify `OCI_DEFAULT`.

## 35.3.9 OCILCRRowStmtWithBindVarGet()

Returns the generated SQL statement, which uses bind variables for column values.

### Purpose

The values for the bind variables are returned separately in arrays. You must preallocate the memory for `sql_stmt` and the arrays, `*sql_stmt_len` must be set to the size of the allocated buffer, and `array_size` must be the length of the arrays. The actual column values in `bind_var_valuesp` points to the values inside the LCR, so it is a shallow copy. If `array_size` is not large enough to hold all the variables, or if `*sql_stmt_len` is not large enough to hold the generated SQL statement, then an error is raised.

### Syntax

```
sword OCILCRRowStmtWithBindVarGet ( OCISvcCtx    *svchp,
                                    OCIError     *errhp,
                                    oratext      *row_stmt,
                                    ub4          *row_stmt_len,
                                    ub2          *num_bind_var,
                                    ub2          *bind_var_dtyp,
                                    void         **bind_var_valuesp,
                                    OCIInd       *bind_var_indp,
                                    ub2          *bind_var_alensp,
                                    ub1          *bind_var_csetidp,
                                    ub1          *bind_var_csetfp,
                                    void         *row_lcrp,
                                    oratext     **chunk_column_names,
                                    ub2          *chunk_column_namesl,
                                    oraub8       *chunk_column_flags,
                                    ub2           array_size,
                                    oratext      *bind_var_syntax,
                                    ub4           mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**row_stmt (IN/OUT)**
The generated SQL statement for the row LCR.

**row_stmt_len (IN/OUT)**
Set to the size of the allocated buffer for `row_stmt` when passed in; returns the length of `row_stmt`.

**num_bind_var (OUT)**
The number of bind variables.

**bind_var_dtyp (IN/OUT)**
Array of data types for the bind variables.

**bind_var_valuesp (IN/OUT)**
Array of values for the bind variables.

**bind_var_indp (IN/OUT)**
Array of `NULL` indicators for the bind variables.

**bind_var_alensp (IN/OUT)**
Array of lengths for the bind variable values.

**bind_var_csetidp (IN/OUT)**
Array of character set IDs for the bind variables.

**bind_var_csetfp (IN/OUT)**
Array of character set forms for the bind variables.

**row_lcrp (IN)**
Pointer to row LCR.

**chunk_column_names (OUT)**
Array of LOB column names in LCR.

**chunk_column_namesl (OUT)**
Array of LOB column name lengths.

**chunk_column_flags (OUT)**
Array of LOB column flags. Possible flags are listed in Comments.

**array_size (IN)**
Size of each of the parameter arrays.

**bind_var_syntax (IN)**
Either (`:`) (binds are of the form `:1`, `:2`, and so on.) or (?) (binds are of the form (?)).

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

The following LCR column flags can be combined using bitwise `OR` operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */
#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */
#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */

/* OCI_LCR_COLUMN_UPDATED is set only for the modified columns in the NEW
 * column list of an update LCR.
 */
```

## 35.3.10 OCILCRNew()

Constructs a new Streams LCR object of the specified type (ROW or DDL) for the given duration.

### Purpose

Constructs a new Streams LCR object of the specified type (ROW or DDL) for the given duration.

### Syntax

```
sword OCILCRNew ( OCISvcCtx     *svchp,
                  OCIError      *errhp,
                  OCIDuration    duration,
                  ub1            lcrtype,
                  void          **lcrp,
                  ub4            mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**duration (IN)**
Memory for the LCR is allocated for this specified duration.

**lcrtype (IN)**
LCR type. Values are:

```
#define OCI_LCR_XROW
#define OCI_LCR_XDDL
```

**lcrp (IN/OUT)**
If `*lcrp` is not `NULL`, an error is raised.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

Note:

- After creation, you are not allowed to change the type of the LCR (ROW or DDL) or duration of the memory allocation.
- Use `OCILCRHeaderSet()` to populate common header fields for row or DDL LCR.
- After the LCR header is initialized, use `OCILCRRowColumnInfoSet()` or `OCILCRDDLInfoSet()` to populate operation specific elements. Use `OCILCRExtraAttributesSet()` to populate extra attribute information.
- Use `OCILCRFree()` to free the LCR created by this function.

## 35.3.11 OCILCRRowColumnInfoGet()

Returns the column fields in a row LCR.

### Purpose

Returns the column fields in a row LCR.

### Syntax

```
sword OCILCRRowColumnInfoGet ( OCISvcCtx    *svchp,
                               OCIError     *errhp,
                               ub2           column_value_type,
                               ub2          *num_columns,
                               oratext     **column_names,
                               ub2          *column_name_lens,
                               ub2          *column_dtyp,
                               void        **column_valuesp,
                               OCIInd       *column_indp,
                               ub2          *column_alensp,
                               ub1          *column_csetfp,
                               oraub8       *column_flags,
                               ub2          *column_csid,
                               void         *row_lcrp,
                               ub2           array_size,
                               ub4           mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**column_value_type (IN)**
ROW LCR column value type; either of:

```
#define OCI_LCR_ROW_COLVAL_OLD
#define OCI_LCR_ROW_COLVAL_NEW
```

**num_columns (OUT)**
Number of columns in the specified column array.

**column_names (OUT)**
An array of column name pointers.

**column_name_lens (OUT)**
An array of column name lengths.

**column_dtyp (OUT)**
An array of column data types. Optional. Data is not returned if `column_dtyp` is `NULL`.

**column_valuesp (OUT)**
An array of column data pointers.

**column_indp (OUT)**
An array of indicators.

**column_alensp (OUT)**
An array of column lengths. Each returned element is the length in bytes.

**column_csetfp (OUT)**
An array of character set forms for the columns. Optional. Data is not returned if the argument is `NULL`.

**column_flags (OUT)**
An array of column flags. Optional. Data is not returned if the argument is `NULL`. See Comments for the values.

**column_csid (OUT)**
An array of character set IDs for the columns.

**row_lcrp (IN)**
`row_lcrp` cannot be `NULL`.

**array_size (IN)**
Size of each of the parameter arrays. An error is returned if `array_size` is less than the number of columns in the requested column list. The actual size of the requested column list is returned through the `num_columns` parameter.

**mode (IN)**
`OCILCR_NEW_ONLY_MODE` - If this mode is specified, then the `new_columns` returned is the count of the columns in the `NEW` column list only. Otherwise, the `new_columns` returned is the number of distinct columns present in either the `NEW` or the `OLD` column list of the given row LCR.

### Comments

- For `INSERT`, this function must only be called to get the NEW column values.
- For `DELETE`, this function must only be called to get the OLD column values.
- For `UPDATE`, this function can be called twice, once to get the NEW column values and once to get the OLD column values.
- This function must not be called for `COMMIT` operations.

The following LCR column flags can be combined using bitwise `OR` operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */

#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */

#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */

/* OCI_LCR_COLUMN_UPDATED is set only for the modified columns in the NEW
 * column list of an update LCR.
 */
```

Table 35-3 lists the currently supported table column data types. For each data type, it lists the corresponding LCR column data type, the C program variable type to cast the LCR column value, and the OCI functions that can manipulate the column values returned from `OCILCRRowColumnInfoGet()`.

**Table 35-3 Table Column Data Types**

| Table Column Data Types | LCR Column Data Type | Program Variable | Conversion Function |
|---|---|---|---|
| VARCHAR, NVARCHAR2 | SQLT_CHR | char * | |
| NUMBER | SQLT_VNU | OCINumber | OCINumberToInt()<br>OCINumberToReal()<br>OCINumberToText() |
| DATE | SQLT_ODT | OCIDate | OCIDateToText()<br>Can access structure directly to get date and time fields. |
| RAW | SQLT_BIN | unsigned char * | |
| CHAR, NCHAR | SQL_AFC | char * | |
| BINARY_FLOAT | SQLT_BFLOAT | float | |
| BINARY_DOUBLE | SQLT_BDOUBLE | double | |
| TIMESTAMP | SQLT_TIMESTAMP | OCIDateTime * | OCIDateTimeGetTime()<br>OCIDateTimeGetDate()<br>OCIDateTimeGetTimeZoneOffset()<br>OCIDateTimeToText() |
| TIMESTAMP WITH TIME ZONE | SQLT_TIMESTAMP_TZ | OCIDateTime * | OCIDateTimeGetTime()<br>OCIDateTimeGetDate()<br>OCIDateTimeGetTimeZoneOffset()<br>OCIDateTimeToText() |

**Table 35-3 (Cont.) Table Column Data Types**

| Table Column Data Types | LCR Column Data Type | Program Variable | Conversion Function |
|---|---|---|---|
| `TIMESTAMP WITH LOCAL TIME ZONE` | `SQLT_TIMESTAMP_LTZ` | `OCIDateTime *` | `OCIDateTimeGetTime()`<br>`OCIDateTimeGetDate()`<br>`OCIDateTimeGetTimeZoneOffset()`<br>`OCIDateTimeToText()` |
| `INTERVAL YEAR TO MONTH` | `SQLT_INTERVAL_YM` | `OCIInterval *` | `OCIIntervalToText()`<br>`OCIIntervalGetYearMonth()` |
| `INTERVAL DAY TO SECOND` | `SQLT_INTERVAL_DS` | `OCIInterval *` | `OCIIntervalToText()`<br>`OCIIntervalGetDaySecond()` |
| `UROWID` | `SQLT_RDD` | `OCIRowid *` | `OCIRowidToChar()` |
| `CLOB` | `SQLT_CHR` or `SQLT_BIN` | `unsigned char *` | [1] |
| `NCLOB` | `SQLT_BIN` | `unsigned char *` | [1] |
| `BLOB` | `SQLT_BIN` | `unsigned char *` | [1] |
| `LONG` | `SQLT_CHR` | `char *` | [1] |
| `LONG RAW` | `SQLT_BIN` | `unsigned char *` | [1] |
| `XMLType` | `SQLT_CHR` or `SQLT_BIN` | `unsigned char *` | |

[1] Call OCIXStreamOutChunkReceive() to get column data.

## 35.3.12 OCILCRRowColumnInfoSet()

Populates column fields in a row LCR.

### Purpose

Populates column fields in a row LCR.

### Syntax

```
sword OCILCRRowColumnInfoSet ( OCISvcCtx    *svchp,
                               OCIError     *errhp,
                               ub2           column_value_type,
                               ub2           num_columns,
                               oratext     **column_names,
                               ub2          *column_name_lens,
                               ub2          *column_dtyp,
                               void        **column_valuesp,
                               OCIInd       *column_indp,
                               ub2          *column_alensp,
                               ub1          *column_csetfp,
                               oraub8       *column_flags,
                               ub2          *column_csid,
                               void         *row_lcrp,
                               ub4           mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**column_value_type (IN)**
ROW LCR Column value types:

```
#define OCI_LCR_ROW_COLVAL_OLD
#define OCI_LCR_ROW_COLVAL_NEW
```

**num_columns (IN)**
Number of columns in each of the array parameters.

**column_names (IN)**
Pointer to an array of column names. Column names must be canonicalized. Column names must follow Oracle Database naming conventions and size limitations.

**column_name_lens (IN)**
Pointer to an array of column name lengths.

**column_dtyp (IN)**
Pointer to an array of column data types. See Comments for valid data types.

**column_valuesp (IN)**
Pointer to an array of column data pointers.

**column_indp (IN)**
Pointer to an indicator array. For all data types, this is a pointer to an array of `OCIInd` values (`OCI_IND_NULL` or `OCI_IND_NOTNULL`).

**column_alensp (IN)**
Pointer to an array of actual column lengths in bytes.

**column_csetfp (IN)**
Pointer to an array of character set forms for the columns. The default form is `SQLCS_IMPLICIT`. Setting this attribute causes the database or national character set to be used on the client side. Set this attribute to `SQLCS_NCHAR` for the national character set or `SQLCS_IMPLICIT` for the database character set. Pass 0 for non-character columns.

**column_flags (IN)**
Pointer to an array of column flags. (See Comments for the list of valid LCR column flags.)

**column_csid (IN)**
Pointer to an array of character set IDs for the columns.

**row_lcrp (IN/OUT)**
`row_lcrp` cannot be `NULL`.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

Note:

- For `INSERT`, this function must only be called to specify the NEW column values.
- For `DELETE`, this function must only be called to specify the OLD column values.
- For `UPDATE`, this function can be called twice, once to specify the NEW column values and once to specify the OLD column values.
- This function must not be called for `COMMIT` operations.

The following LCR column flags can be combined using the bitwise `OR` operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */

#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */

#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */

/* OCI_LCR_COLUMN_UPDATED is set only for the modified columns in the NEW
 * column list of an update LCR.
 */
```

Valid data types are:

```
SQLT_AFC           SQLT_TIMESTAMP
SQLT_DAT           SQLT_TIMESTAMP_TZ
SQLT_BFLOAT        SQLT_TIMESTAMP_LTZ
SQLT_BDOUBLE       SQLT_INTERVAL_YM
SQLT_NUM           SQLT_INTERVAL_DS
SQLT_VCS
SQLT_ODT
SQLT_INT
SQLT_BIN
SQLT_CHR
SQLT_RDD
SQLT_VST
SQLT_INT
SQLT_FLT
```

## 35.3.13 OCILCRDDLInfoSet()

Populates DDL-specific fields in a DDL LCR.

### Purpose

Populates DDL-specific fields in a DDL LCR.

### Syntax

```
sword OCILCRDDLInfoSet ( OCISvcCtx    *svchp,
                         OCIError     *errhp,
                         oratext      *object_type,
                         ub2           object_type_len,
                         oratext      *ddl_text,
                         ub4           ddl_text_len,
                         oratext      *logon_user,
                         ub2           logon_user_len,
                         oratext      *current_schema,
                         ub2           current_schema_len,
                         oratext      *base_table_owner,
                         ub2           base_table_owner_len,
                         oratext      *base_table_name,
                         ub2           base_table_name_len,
                         oraub8        flag,
                         void         *ddl_lcrp,
                         ub4           mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**object_type (IN)**
The type of object on which the DDL statement was executed. See Comments for the valid object types.

**object_type_len (IN)**
Length of the `object_type` string without the `NULL` terminator.

**ddl_text (IN)**
The text of the DDL statement. This parameter must be set to a non-`NULL` value. DDL text must be in Oracle Database DDL format.

**ddl_text_len (IN)**
DDL text length in bytes without the `NULL` terminator.

**logon_user (IN)**
Canonicalized name of the user whose session executed the DDL statement.

**logon_user_len (IN)**
Length of the `logon_user` string without the `NULL` terminator. Must follow Oracle Database naming conventions and size limitations.

**current_schema (IN)**
The canonicalized schema name that is used if no schema is specified explicitly for the modified database objects in `ddl_text`. If a schema is specified in `ddl_text` that differs from the one specified for `current_schema`, then the function uses the schema specified in `ddl_text`.
This parameter must be set to a non-`NULL` value.

**current_schema_len (IN)**
Length of the `current_schema` string without the `NULL` terminator. Must follow Oracle Database naming conventions and size limitations.

**base_table_owner (IN)**
If the DDL statement is a table-related DDL (such as `CREATE TABLE` or `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_owner` specifies the canonicalized owner of the table involved. Otherwise, `base_table_owner` is `NULL`.

**base_table_owner_len (IN)**
Length of the `base_table_owner` string without the `NULL` terminator. Must follow Oracle Database naming conventions and size limitations.

**base_table_name (IN)**
If the DDL statement is a table-related DDL (such as `CREATE TABLE` or `ALTER TABLE`), or if the DDL statement involves a table (such as creating a trigger on a table), then `base_table_name` specifies the canonicalized name of the table involved. Otherwise, `base_table_name` is `NULL`.

**base_table_name_len (IN)**
Length of the `base_table_name` without the `NULL` terminator. Must follow Oracle Database naming conventions and size limitations.

**flag (IN)**
DDL LCR flag. (Not currently used; used for future extension.) Specify `OCI_DEFAULT`.

**ddl_lcrp (IN/OUT)**
`ddl_lcrp` cannot be `NULL`.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

The following are valid object types:

```
CLUSTER
FUNCTION
INDEX
OUTLINE
PACKAGE
PACKAGE BODY
PROCEDURE
SEQUENCE
SYNONYM
TABLE
TRIGGER
TYPE
USER
VIEW
```

`NULL` is also a valid object type. Specify `NULL` for all object types not listed.

## 35.3.14 OCILCRGetLCRIDVersion()

Determines the LCRID version of a given position.

### Purpose

Determines the LCRID version of a given position.

### Syntax

```
sword OCILCRGetLCRIDVersion(OCISvcCtx   *svchp,
                            OCIError    *errhp,
                            ub1         *position,
                            ub2          position_len,
                            ub1         *version);
```

### Parameters

**svchp (IN)**
The OCI service context.

**errhp (IN)**
The OCI Error handle.

**position (IN)**
The position.

**position_len (IN)**
The length of `position`.

**version (OUT)**
The LCRID version for the given input `position`. A value of `1` should be specified for the original `OCI_LCRID_V1` format. A value of `2` should be specified for the `OCI_LCRID_V2` format.

```
#define  OCI_LCRID_V1       1
#define  OCI_LCRID_V2       2
```

### Returns

`OCI_SUCCESS` if the LCRID version is valid, `OCI_ERROR` otherwise.

### Usage Notes

None.

## 35.3.15 OCILCRHeaderSet()

Initializes the common header fields for ROW or DDL LCR.

### Purpose

Initializes the common header fields for row or DDL LCR.

### Syntax

```
sword OCILCRHeaderSet ( OCISvcCtx  *svchp,
                        OCIError   *errhp,
                        oratext    *src_db_name,
                        ub2         src_db_name_len,
                        oratext    *cmd_type,
                        ub2         cmd_type_len,
                        oratext    *owner,
                        ub2         owner_len,
                        oratext    *oname,
                        ub2         oname_len,
                        ub1        *tag,
                        ub2         tag_len,
                        oratext    *txid,
                        ub2         txid_len,
                        OCIDate    *src_time,
                        ub1        *position,
                        ub2         position_len,
                        oraub8      flag,
                        void       *lcrp,
                        ub4         mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**src_db_name (IN)**
Canonicalized source database name. Must be non-`NULL`.

**src_db_name_len (IN)**
Length of the `src_db_name` string in bytes excluding the `NULL` terminator. Must follow Oracle Database naming conventions and size limitations.

**cmd_type (IN)**
For row LCRs: One of the following values:

> **Note:**
>
> The values, `#define OCI_LCR_ROW_CMD_ROLLBACK` and `#define OCI_LCR_ROW_CMD_START_TX`, are available starting with Oracle Database 11*g* Release 2 (11.2.0.2).

```
#define OCI_LCR_ROW_CMD_INSERT
#define OCI_LCR_ROW_CMD_DELETE
#define OCI_LCR_ROW_CMD_UPDATE
#define OCI_LCR_ROW_CMD_COMMIT
#define OCI_LCR_ROW_CMD_ROLLBACK
#define OCI_LCR_ROW_CMD_START_TX
```

```
#define OCI_LCR_ROW_CMD_LOB_WRITE
#define OCI_LCR_ROW_CMD_LOB_TRIM
#define OCI_LCR_ROW_CMD_LOB_ERASE
```

For DDL LCRs: One of the command types in OCI_ATTR_SQLFNCODE.

**cmd_type_len (IN)**
Length of `cmd_type`.

**owner (IN)**
Canonicalized table owner name. Owner is not required for `COMMIT` LCR.

**owner_len (IN)**
Length of the `owner` string in bytes excluding the `NULL` terminator. Must follow Oracle Database naming conventions and size limitations.

**oname (IN)**
Canonicalized table name. Not required for `COMMIT` LCR.

**oname_len (IN)**
Length of the `oname` string in bytes excluding the `NULL` terminator. Must follow Oracle Database naming conventions and size limitations.

**tag (IN)**
A binary tag that enables tracking of the LCR. For example, you can use this tag to determine the original source database of the DML statement if apply forwarding is used.

**tag_len (IN)**
Number of bytes in the tag. Cannot exceed 2000 bytes.

**txid (IN)**
Transaction ID. Must be non-`NULL`.

**txid_len (IN)**
Length of the `txid` string in bytes, excluding the `NULL` terminator. Must follow Oracle Database naming conventions and size limitations.

**src_time (IN)**
The time when the change was generated in the online redo log file of the source database.

**position (IN)**
Position for LCR. Must be non-`NULL` and byte-comparable.

**position_len (IN)**
Length of position. Must be greater than zero.

**flag (IN)**
LCR flag. Possible flags are listed in Comments.

**lcrp (IN/OUT)**
`lcrp` cannot be `NULL`.

**mode (IN)**
Specify `OCI_DEFAULT`.

**Comments**

Note:

- This function sets all internal fields of the LCR to `NULL` including extra attributes.
- This function *does not* deep copy the passed-in values. You must ensure data is valid for the duration of the LCR.
- For `COMMIT` LCRs, `owner` and `oname` information are not required. Provide valid values for `src_db_name`, `cmd_type`, `tag`, `txid`, and `position`.
- For ROW LCRs, use `OCILCRRowColumnInfoSet()` to populate row LCR-specific column information.
- For DDL LCRs, use `OCILCRDDLInfoSet()` to populate DDL operation specific information.
- For ROW or DDL LCRs, use `OCILCRAttributesSet()` to populate extra attribute information.

The following are LCR flags:

```
#define OCI_ROWLCR_HAS_ID_KEY_ONLY  /* only has ID key cols */
#define OCI_ROWLCR_SEQ_LCR          /* sequence lcr */
```

## 35.3.16 OCILCRLobInfoGet()

Returns the LOB information for a piece-wise LOB LCR generated from a `DBMS_LOB` or `OCILob` procedure.

### Purpose

Returns the LOB information for a piece-wise LOB LCR generated from a `DBMS_LOB` or `OCILob` procedure.

### Syntax

```
sword OCILCRLobInfoGet ( OCISvcCtx   *svchp,
                         OCIError    *errhp,
                         oratext    **column_name,
                         ub2         *column_name_len,
                         ub2         *column_dty,
                         oraub8      *column_flag,
                         ub4         *offset,
                         ub4         *size,
                         void        *row_lcrp,
                         ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**column_name (OUT)**
LOB column name.

**column_name_len (OUT)**
Length of LOB column name.

**column_dty (OUT)**
Column data type (either `SQLT_CHR` or `SQLT_BIN`).

**column_flag (OUT)**
Column flag. See Comments in `OCILCRRowColumnInfoSet()`.

**offset (OUT)**
LOB operation offset in code points. Only returned for `LOB WRITE` and `LOB TRIM` operations. This is the same as the `offset` parameter for `OCILobErase()` or the `offset` parameter in `OCILobWrite()`.

**size (OUT)**
LOB operation size in code points. Only returned for `LOB TRIM` and `LOB ERASE` operations. This is the same as the `new_length` parameter in `OCILobTrim()` or the `amtp` parameter in `OCILobErase()`.

**row_lcrp (IN)**
Pointer to a row LCR.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

Returns `OCI_SUCCESS` or `OCI_ERROR`.

### Related Topics

- OCILCRRowColumnInfoSet()
  Populates column fields in a row LCR.

## 35.3.17 OCILCRLobInfoSet()

Sets the LOB information for a piece-wise LOB LCR.

### Purpose

This call is valid when the input LCR is a `LOB_WRITE`, `LOB_ERASE`, or `LOB_TRIM`; otherwise, an error is returned.

### Syntax

```
sword OCILCRLobInfoSet ( OCISvcCtx   *svchp,
                         OCIError    *errhp,
                         oratext     *column_name,
                         ub2          column_name_len,
                         ub2          column_dty,
                         oraub8       column_flag,
                         ub4          offset,
                         ub4          size,
                         void        *row_lcrp,
                         ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**column_name (IN)**
LOB column name.

**column_name_len (IN)**
Length of LOB column name.

**column_dty (IN)**
Column data type (either `SQLT_CHR` or `SQLT_BIN`).

**column_flag (IN)**
Column flag. See Comments in `OCILCRRowColumnInfoSet()`.

**offset (IN)**
LOB operation offset in code points. Only required for `LOB WRITE` and `LOB TRIM` operations. This is the same as the `soffset` parameter for `OCILobErase()` or the `offset` parameter in `OCILobWrite()`.

**size (IN)**
LOB operation size in code points. Only required for `LOB TRIM` and `LOB ERASE` operations.This is the same as the `new_length` parameter in `OCILobTrim()` or the `amtp` parameter in `OCILobErase()`.

**row_lcrp (IN/OUT)**
Pointer to a row LCR.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

Returns `OCI_SUCCESS` or `OCI_ERROR`.

### Related Topics

- OCILCRRowColumnInfoSet()
  Populates column fields in a row LCR.

## 35.3.18 OCILCRSCNsFromPosition()

Returns the SCN and the commit SCN from the position value.

### Purpose

The input position must be one that is obtained from an XStream outbound server. An error is returned if the input position does not conform to the expected format.

#### Syntax

```
sword OCILCRSCNsFromPosition ( OCISvcCtx   *svchp,
                               OCIError    *errhp,
                               ub1         *position,
                               ub2          position_len,
                               OCINumber   *scn,
                               OCINumber   *commit_scn,
                               ub4          mode );
```

#### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**position (IN)**
LCR position value.

**position_len (IN)**
Length of LCR position value.

**scn (OUT)**
SCN number embedded in the given LCR position.

**commit_scn (OUT)**
The commit SCN embedded in the given position.

**mode (IN)**
Mode flags used for future expansion. Specify `OCI_DEFAULT`.

#### Comments

This function can handle both version 1 and version 2 position (LCRID) values.

### 35.3.19 OCILCRSCNToPosition()

Converts an SCN to a position.

#### Purpose

The generated position can be passed as the `last_position` to `OCIXStreamOutAttach()` to filter the LCRs with commit SCN less than the given SCN and the LCR's SCN less than the given SCN. Therefore, the first LCR sent by the outbound server is either:

- A commit LCR at the given SCN, or
- The first LCR of the subsequent transaction with commit SCN greater than or equal to the given SCN.

#### Syntax

```
sword OCILCRSCNToPosition ( OCISvcCtx   *svchp,
                            OCIError    *errhp,
                        ub1        *position,
                        ub2        *position_len,
                        OCINumber  *scn,
                        ub4         mode );
```

**Parameters**

**svchp (IN)**
OCI service context.

**errhp (IN)**
OCI error handle.

**position (OUT)**
The resulting position. You must preallocate `OCI_LCR_MAX_POSITION_LEN` bytes.

**position_len (OUT)**
Length of `position`.

**scn (IN)**
The SCN to be stored in `position`.

**mode (IN)**
Mode flags (Not currently used; used for future extension).

**Comments**

Returns `OCI_SUCCESS` if the conversion succeeds, `OCI_ERROR` otherwise.

This function will return a version 1 LCRID.

An error will be raised if the SCN value is larger than the maximum value.

## 35.3.20 OCILCRScnToPosition2()

Converts SCN to position (LCRID), which can handle both version 1 and version 2.

**Purpose**

Converts SCN to position (LCRID) for either version 1 or version 2.

**Syntax**

```
sword OCILCRSCNToPosition2(OCISvcCtx  *svchp,
                           OCIError   *errhp,
                           ub1        *position,
                           ub2        *position_len,
                           OCINumber  *scn,
                           ub1         version,
                           ub4         mode);
```

**Parameters**

**svchp (IN)**
The OCI service context.

**errhp (IN)**
The OCI Error handle.

**position (OUT)**
The result position value (in the specified version) returned. The memory must be preallocated.

**position_len (OUT)**
The length of `position`.

**scn (IN)**
The SCN value to be stored in `position`.

**version (IN)**
The version to which to convert the LCRID value. A value of `1` should be specified for version 1. A value of `2` should be specified for version 2.

```
#define  OCI_LCRID_V1       1
#define  OCI_LCRID_V2       2
```

**mode (IN)**
The mode flags (Not currently used; used for future extension)..

### Returns

`OCI_SUCCESS` if the conversion is successful, `OCI_ERROR` otherwise.

### Usage Notes

If version `1` is specified, and the SCN value is larger than the maximum value, then an error will be raised.

The given SCN value is assumed to be the commit SCN value.

## 35.3.21 OCILCRWhereClauseGet()

Gets the `WHERE` clause statement for the given row LCR.

### Purpose

Gets the `WHERE` clause statement for the given row LCR.

### Syntax

```
sword OCILCRWhereClauseGet ( OCISvcCtx  *svchp,
                             OCIError   *errhp,
                             oratext    *wc_stmt,
                             ub4        *wc_stmt_len,
                             void       *row_lcrp,
                             ub4         mode );
```

### Parameters

**svchp (IN/OUT)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**wc_stmt (OUT)**
SQL statement equivalent to the LCR.

**wc_stmt_len (IN/OUT)**
Length of the `wc_stmt` buffer.

**row_lcrp (IN)**
Row LCR to be converted to SQL.

**mode (IN)**
Mode flags used for future expansion. Specify `OCI_DEFAULT`.

### Comments

The `WHERE` clause generated for an `INSERT` LCR has all the columns that are being inserted. This `WHERE` clause could be used to identify the inserted row after it is inserted, for example, like "returning `ROWID`".

```
INSERT INTO TAB(COL1) VALUES (10) -> WHERE COL1=10
```

The `WHERE` clause generated for `UPDATE` has all the columns in the old column list. However, the values of the columns are that of the new value if it exists in the new column list of the `UPDATE`. If the column does not have a new value, then the old column value is used.

```
UPDATE TAB SET COL1 = 10 WHERE COL1 = 20 -> WHERE COL1 = 10
UPDATE TAB SET COL2 = 20 WHERE COL1 = 20 -> WHERE COL1 = 20
```

The `WHERE` clause for DELETE uses the columns and values from the old column list.

LOB piecewise operations use the new columns and values for generating the `WHERE` clause.

### Returns

`OCI_SUCCESS` or `OCI_ERROR`.

## 35.3.22 OCILCRWhereClauseWithBindVarGet()

Gets the `WHERE` clause statement with bind variables for the given row LCR.

### Purpose

Gets the `WHERE` clause statement with bind variables for the given row LCR.

### Syntax

```
sword OCILCRWhereClauseWithBindVarGet ( OCISvcCtx  *svchp,
                                        OCIError   *errhp,
                                        oratext    *wc_stmt,
                                        ub4        *wc_stmt_len,
                                        ub2        *num_bind_var,
                                        ub2        *bind_var_dtyp,
                                        void       **bind_var_valuesp,
                                        OCIInd     *bind_var_indp,
                                         ub2         *bind_var_alensp,
                                         ub2         *bind_var_csetidp,
                                         ub1         *bind_var_csetfp,
                                         void        *row_lcrp,
                                         ub2          array_size,
                                         oratext     *bind_var_syntax,
                                         ub4          mode );
```

**Parameters**

**svchp (IN/OUT)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**wc_stmt (OUT)**
SQL statement equivalent to the LCR.

**wc_stmt_len (IN/OUT)**
Length of the `wc_stmt` buffer.

**num_bind_var (OUT)**
Number of bind variables.

**bind_var_dtyp (OUT)**
Array of data types of bind variables.

**bind_var_valuesp (OUT)**
Array of values of bind variables.

**bind_var_indp (OUT)**
Array of `NULL` indicators of bind variables.

**bind_var_alensp (OUT)**
Array of lengths of bind values.

**bind_var_csetidp (OUT)**
Array of char set IDs of binds.

**bind_var_csetfp (OUT)**
Array of char set forms of binds.

**row_lcrp (IN)**
Row LCR to be converted to SQL.

**array_size (IN)**
Size of the array of bind values.

**bind_var_syntax (IN)**
Native syntax to be used for binds.

**mode (IN)**
Mode flags for future expansion. Specify `OCI_DEFAULT`.

### Comments

If `array_size` is not large enough to accommodate the number of columns in the requested column list, then `OCI_ERROR` is returned. The expected `array_size` is returned through the `num_bind_var` parameter.

`bind_var_syntax` for Oracle Database should contain (`:`). This generates positional binds such as `:1`, `:2`, `:3`, and so on. For non-Oracle databases input the string that must be used for binds.

The `WHERE` clause generated for `INSERT` LCR has all the columns that are being inserted. This `WHERE` clause can identify the inserted row after it is inserted, for example, like "returning `ROWID`".

```
INSERT INTO TAB(COL1) VALUES (10) -> WHERE COL1=10
```

The `WHERE` clause generated for `UPDATE` has all the columns in the old column list. However, the values of the columns are that of the new column value of the column if it exists in the new values of the `UPDATE`. If the column appears only in the old column, then the old column value is used.

```
UPDATE TAB SET COL1 = 10 WHERE COL1 = 20 -> WHERE COL1 = 10
UPDATE TAB SET COL2 = 20 WHERE COL1 = 20 -> WHERE COL1 = 20
```

The `WHERE` clause for `DELETE` uses the columns and values from the old column list.

LOB piecewise operations use the new columns and values for generating the `WHERE` clause.

### Returns

`OCI_SUCCESS` or `OCI_ERROR`.

## 35.3.23 OCIXStreamInAttach()

Attaches to an inbound server.

### Purpose

The client application must connect to the database using a dedicated connection.

### Syntax

```
sword OCIXStreamInAttach ( OCISvcCtx  *svchp,
                           OCIError   *errhp,
                           oratext    *server_name,
                           ub2         server_name_len,
                           oratext    *source_name,
                           ub2         source_name_len,
                           ub1        *last_position,
                           ub2        *last_position_len,
                           ub4         mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**server_name (IN)**
XStream inbound server name.

**server_name_len (IN)**
Length of the XStream inbound server name.

**source_name (IN)**
Source name to identify the data source.

**source_name_len (IN)**
Source name length.

**last_position (OUT)**
Last position received by inbound server. Optional. If specified, then you must preallocate `OCI_LCR_MAX_POSITION_LEN` bytes for the return value.

**last_position_len (OUT)**
Length of `last_position`. Must be non-`NULL` if `last_position` is non-`NULL`.

**mode (IN)**
`OCIXSTREAM_IN_ATTACH_RESTART_INBOUND` - If this mode is specified, then this function can notify the server to restart the inbound server regardless of whether it is in a disabled or aborted state. If you do not pass in this mode and the inbound server is in an aborted state when this call is made, then the function returns an error.

### Comments

The name of the inbound server must be provided because multiple inbound servers can be configured in one Oracle instance. This function returns `OCI_ERROR` if any error is encountered while attaching to the inbound server. Only one client can attach to an XStream inbound server at any time. An error is returned if multiple clients attempt to attach to the same inbound server or if the same client attempts to attach to multiple inbound servers concurrently.

This function returns the last position received by the inbound server. Having successfully attached to the server, the client should resume sending LCRs with positions greater than this `last_position` since the inbound server discards all LCRs with positions less than or equal to the `last_position`.

Returns either `OCI_SUCCESS` or `OCI_ERROR` status code.

## 35.3.24 OCIXStreamInDetach()

Detaches from the inbound server.

### Purpose

Detaches from the inbound server.

### Syntax

```
sword OCIXStreamInDetach ( OCISvcCtx  *svchp,
                           OCIError   *errhp,
                           ub1        *processed_low_position,
                             ub2        *processed_low_position_len,
                             ub4         mode );
```

**Parameters**

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**processed_low_position (OUT)**
The server's processed low position.

**processed_low_position_len (OUT)**
Length of `processed_low_position`.

**mode (IN)**
Specify `OCI_DEFAULT`.

**Comments**

You must pass in a preallocated buffer for the position argument. The maximum length of this buffer is `OCI_LCR_MAX_POSITION_LEN`. This position is exposed in `DBA_XSTREAM_INBOUND_PROGRESS` view

This call returns the server's processed low position. If this function is invoked while a `OCIXStreamInLCRSend()` call is in progress, then it immediately terminates that call before detaching from the inbound server.

Returns either `OCI_SUCCESS` or `OCI_ERROR` status code.

## 35.3.25 OCIXStreamInLCRSend()

Sends an LCR stream from the client to the attached inbound server.

**Purpose**

To avoid a network round-trip for every `OCIXStreamInLCRSend()` call, the connection is tied to this call and terminates the call after an ACK interval since the LCR stream is initiated to the server.

**Syntax**

```
sword OCIXStreamInLCRSend ( OCISvcCtx   *svchp,
                            OCIError    *errhp,
                            void        *lcrp,
                            ub1          lcrtype,
                            oraub8       flag,
                            ub4          mode );
```

**Parameters**

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**lcrp (IN)**
Pointer to the new LCR to send. It cannot be `NULL`.

**lcrtype (IN)**
LCR type. Either of:

```
#define OCI_LCR_XROW
#define OCI_LCR_XDDL
```

**flag (IN)**
If bit `OCI_XSTREAM_MORE_ROW_DATA` (0x01) is set, then LCR contains more chunk data. You must call `OCIXStreamInChunkSend()` before calling `OCIXStreamInLCRSend()` again.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

Return codes are:

- `OCI_STILL_EXECUTING` means that the current call is still in progress. The connection associated with the specified service context handle is still tied to this call for streaming the LCRs to the server. An error is returned if you attempt to use the same connection to execute any OCI calls that require database round-trip, for example, `OCIStmtExecute()`, `OCIStmtFetch()`, `OCILobRead()`, and so on. `OCILCR*` calls are local calls; thus, they are valid while this call is in progress.
- `OCI_SUCCESS` means the current call is completed. You can execute `OCIStmt*`, `OCILob*`, and so on from the same service context.
- `OCI_ERROR` means this call encounters some errors. Use `OCIErrorGet()` to obtain information about the error.

### Related Topics

- Server Handle Attributes
  Lists and describes server handle attributes.

## 35.3.26 OCIXStreamInLCRCallbackSend()

Sends an LCR stream to the attached inbound server.

### Purpose

You must specify a callback to construct each LCR for streaming. If some LCRs contain chunk data, then a second callback must be provided to create each chunk data.

### Syntax

```
sword OCIXStreamInLCRCallbackSend (
          OCISvcCtx                     *svchp,
          OCIError                      *errhp,
          OCICallbackXStreamInLCRCreate  createlcr_cb,
          OCICallbackXStreamInChunkCreate  createchunk_cb,
          void                            *usrctxp,
          ub4                              mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**createlcr_cb (IN)**
Client callback procedure to be invoked to generate an LCR for streaming. Cannot be `NULL`.

**createchunk_cb (IN)**
Client callback procedure to be invoked to create each chunk. Can be `NULL` if you do not need to send any LCR with LOB or `LONG` or `XMLType` columns. `OCI_ERROR` is returned if this argument is `NULL` and you attempt to send an LCR with additional chunk data.

**usrctxp (IN)**
User context to pass to both callback functions.

**mode (IN)**
Specify `OCI_DEFAULT` fore now.

### Comments

Return code: `OCI_ERROR` or `OCI_SUCCESS`.

The `createlcr_cb` argument must be of type `OCICallbackXStreamInLCRCreate`:

```
typedef sb4  (*OCICallbackXStreamInLCRCreate)
              void  *usrctxp, void **lcrp, ub1 *lcrtyp, oraub8 *flag);
```

Parameters of `OCICallbackXStreamInLCRCreate()`:

**usrctxp (IN/OUT)**
Pointer to the user context.

**lcrp (OUT)**
Pointer to the LCR to be sent.

**lcrtyp (OUT)**
LCR type (`OCI_LCR_XROW` or `OCI_LCR_XDDL`).

**flag (OUT)**
If `OCI_XSTREAM_MORE_ROW_DATA` is set, then the current LCR has more chunk data.

The input parameter to the callback is the user context. The output parameters are the new LCR, its type, and a flag. If the generated LCR contains additional chunk data, then this flag must have the `OCI_XSTREAM_MORE_ROW_DATA` (0x01) bit set. The valid return codes from the `OCICallbackXStreamInLCRCreate()` callback function are `OCI_CONTINUE` or `OCI_SUCCESS`. This callback function must return `OCI_CONTINUE` to continue processing the `OCIXStreamInLCRCallbackSend()` call. Any return code other than `OCI_CONTINUE` signals that the client wants to terminate the `OCIXStreamInLCRCallbackSend()` call immediately. In addition, a `NULL` LCR returned from the `OCICallbackXStreamInLCRCreate()` callback function signals that the client wants to terminate the current call.

The `createchunk_cb` argument must be of type `OCICallbackXStreamInChunkCreate`:

```
typedef sb4 (*OCICallbackXStreamInChunkCreate)
void     *usrctxp,
oratext  **column_name,
ub2      *column_name_len,
ub2      *column_dty,
oraub8   *column_flag,
ub2      *column_csid,
ub4      *chunk_bytes,
ub1      **chunk_data,
oraub8   *flag);
```

The input parameters of the `createchunk_cb()` procedure are the user context and the information about the chunk.

Parameters of `OCICallbackXStreamInChunkCreate()`:

**usrctxp (IN/OUT)**
Pointer to the user context.

**column_name (OUT)**
Column name of the current chunk.

**column_name_len (OUT)**
Length of the column name.

**column_name_dty (OUT)**
Chunk data type (`SQLT_CHR` or `SQLT_BIN`).

**column_flag (OUT)**
See Comments in `OCIXStreamInChunkSend()`.

**column_csid (OUT)**
Column character set ID. Relevant only if the column is an `XMLType` column (that is, `column_flag` has the `OCI_LCR_COLUMN_XML_DATA` bit set).

**chunk_bytes (OUT)**
Chunk data length in bytes.

**chunk_data (OUT)**
Chunk data pointer.

**flag (OUT)**
If `OCI_XSTREAM_MORE_ROW_DATA` is set, then the current LCR has more chunk data.

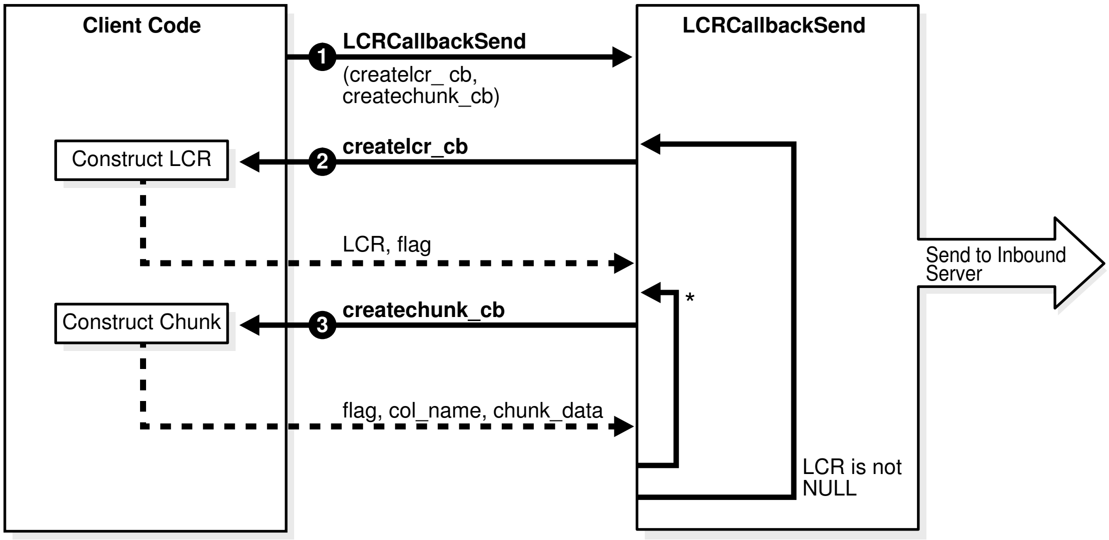
The `OCIXStreamInLCRCallbackSend()` function invokes the `createlcr_cb()` procedure to obtain each LCR to send to the server. If the return flag from the `createlcr_cb()` procedure has the `OCI_XSTREAM_MORE_ROW_DATA` bit set, then it invokes the `createchunk_cb()` procedure to obtain each chunk. It repeatedly calls the `createchunk_cb()` procedure while the flag returned from this callback has the `OCI_XSTREAM_MORE_ROW_DATA` bit set. When this bit is not set, this function cycles back to invoke the `createlcr_cb()` procedure to get the next LCR. This cycle is repeated until the `createlcr_cb()` procedure returns a `NULL` LCR or when at the transaction boundary after an ACK interval has elapsed since the call began.

The valid return codes from the `OCICallbackXStreamInChunkCreate()` callback function are `OCI_CONTINUE` or `OCI_SUCCESS`. This callback function must return `OCI_CONTINUE` to continue processing the `OCIXStreamInLCRCallbackSend()` call. Any return code other than `OCI_CONTINUE` signals that the client wants to terminate the `OCIXStreamInLCRCallbackSend()` call immediately.

Because terminating the current call flushes the network and incurs another network round-trip in the next call, you must avoid returning a `NULL` LCR immediately when there is no LCR to send. Doing this can greatly reduce network throughput and affect performance. During short idle periods, you can add some delays in the callback procedure instead of returning a `NULL` LCR immediately to avoid flushing the network too frequently.

Figure 35-1 shows the execution flow of the `OCIXStreamInLCRCallbackSend()` function.

**Figure 35-1 Execution Flow of the OCIXStreamInLCRCallbackSend() Function**



* `While OCI_XSTREAM_MORE_ROW_DATA` is set

Description of Figure 35-1:

- At 1, the user invokes the `OCIXStreamInLCRCallbackSend()` providing two callbacks. This function initiates an LCR inbound stream to the server.
- At 2, this function invokes the `createlcr_cb()` procedure to get an LCR from the callback to send to the server. If the return LCR is `NULL`, then this function exits.
- If the flag from 2 indicates the current LCR has more data (that is, the `OCI_XSTREAM_MORE_ROW_DATA` bit is set), then this function proceeds to 3; otherwise, it loops back to 2 to get the next LCR.
- At 3, this function invokes `createchunk_cb()` to get the chunk data to send to the server. If the flag from this callback has the `OCI_XSTREAM_MORE_ROW_DATA` bit set, then it repeats 3; otherwise, it loops back to 2 to get the next LCR from the user. If

any callback function returns any values other than `OCI_CONTINUE`, then the `OCIXStreamInLCRCallbackSend()` call terminates.

Following is a sample client pseudocode snippet for callback mode (error checking is not included for simplicity):

```
main
{
   /* Attach to inbound server */
   OCIXStreamInAttach();

   /* Get the server's processed low position to determine
    * the position of the first LCR to generate.
    */
   OCIXStreamInProcessedLWMGet(&lwm);

   while (TRUE)
   {
      /* Initiate LCR inbound stream */
      OCIXStreamInLCRCallbackSend(createlcr_cb, createchunk_cb);

      OCIXStreamInProcessedLWMGet(&lwm);

      if (some terminating condition)
         break;
   }
   OCIXStreamInDetach(&lwm);
}


createlcr_cb (IN usrctx, OUT lcr, OUT flag)
{
   if (have more LCRs to send)
   {
      /* construct lcr */
      OCILCRHeaderSet(lcr);
      OCILCRRowColumnInfoSet(lcr);

      if (lcr has LOB | LONG | XMLType columns)
         Set OCI_XSTREAM_MORE_ROW_DATA flag;

      if (lcr is LOB_ERASE | LOB_TRIM | LOB_WRITE)
         OCILCRLobInfoSet(lcr);
   }
   else if (idle timeout expires)
   {
      lcr = null;
   }
}

createchunk_cb (IN usrctx, OUT chunk, OUT flag)
{
   /* set col_name, col_flag, chunk data, and so on */
   construct_chunk;

   if (last chunk of current column)
   {
      set col_flag |= OCI_LCR_COLUMN_LAST_CHUNK;

      if (last column)
         clear OCI_XSTREAM_MORE_ROW_DATA flag;
    }
}
```

**Related Topics**

- OCIXStreamInChunkSend()
  Sends a chunk to the inbound server.

## 35.3.27 OCIXStreamInProcessedLWMGet()

Gets the local processed low position that is cached at the client.

### Purpose

This function can be called anytime while the client is attached to an XStream inbound server. Clients, using the callback mode to stream LCRs to the server (see `OCIXStreamInLCRCallbackSend()`), can invoke this function while in the callback procedures.

### Syntax

```
sword OCIXStreamInProcessedLWMGet ( OCISvcCtx  *svchp,
                                    OCIError   *errhp,
                                    ub1        *processed_low_position,
                                    ub2        *processed_low_position_len,
                                    ub4         mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**processed_low_position (OUT)**
The processed low position maintained at the client.

**processed_low_position_len (OUT)**
Length of `processed_low_position`.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

After attaching to an XStream inbound server, a local copy of the server's processed low position (see `OCIXStreamOutProcessedLWMSet()`) is cached at the client. This local copy is refreshed with the server's low position when each of the following calls returns `OCI_SUCCESS`:

- `OCIXStreamInAttach()`
- `OCIXStreamInLCRSend()`
- `OCIXStreamInLCRCallbackSend()`
- `OCIXStreamInFlush()`

Return code: `OCI_ERROR` or `OCI_SUCCESS`.

You must pass in a preallocated buffer for the position argument. The maximum length of this buffer is `OCI_LCR_MAX_POSITION_LEN`. This position is exposed in the `DBA_XSTREAM_INBOUND_PROGRESS` view.

The client can use this position to periodically purge the logs used to generate the LCRs at or below this position.

**Related Topics**

- OCIXStreamInLCRCallbackSend()
  Sends an LCR stream to the attached inbound server.
- OCIXStreamOutProcessedLWMSet()
  Updates the local copy of the processed low position.

## 35.3.28 OCIXStreamInErrorGet()

Returns the first error encountered by the inbound server since attaching to it.

### Purpose

Returns the first error encountered by the inbound server since the `OCIXStreamInAttach()` call.

### Syntax

```
sword OCIXStreamInErrorGet ( OCISvcCtx *svchp,
                             OCIError  *errhp,
                             sb4       *errcodep,
                             oratext   *msgbuf,
                             ub2        msg_bufsize,
                             ub2       *msg_len,
                             oratext   *txn_id,
                             ub2        txn_id_bufsize,
                             ub2       *txn_id_len );
```

### Parameters

**svchp (IN/OUT)**
OCI service handle.

**errhp (IN/OUT)**
Error Handle.

**errcodep (OUT)**
Error code.

**msgbuf (IN/OUT)**
Preallocated message buffer.

**msg_bufsize (IN)**
Message buffer size.

**msg_len (OUT)**
Length of returned error message.

**txn_id (IN/OUT)**
Preallocated transaction ID buffer.

**txn_id_bufsize (IN)**
The transaction ID buffer size.

**txn_id_len (OUT)**
Length of the returned transaction ID.

### Comments

The maximum size for the returned transaction ID is `OCI_LCR_MAX_TXID_LEN`. If the allocated buffer for `txn_id` is too small, then this routine returns `ORA-29258`. The maximum size for the returned error msg is `OCI_ERROR_MAXMSG_SIZE`. If the allocated size for `msgbuf` is too small, then the returned message is truncated.

## 35.3.29 OCIXStreamInFlush()

Used to flush the network while attaching to an XStream inbound server.

### Purpose

It terminates any in-progress `OCIXStreamInLCRSend()` call associated with the specified service context.

### Syntax

```
sword OCIXStreamInFlush ( OCISvcCtx    *svchp,
                          OCIError     *errhp,
                          ub4           mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**mode (IN)**
`OCIXSTREAM_IN_FLUSH_WAIT_FOR_COMPLETE` - If this mode is specified, then this function flushes the network, and then waits for all complete and rollback transactions that have been sent to the inbound server to complete before returning control to the client.

### Comments

Return code: `OCI_ERROR` or `OCI_SUCCESS`.

Each call incurs a database round-trip to get the server's processed low position, which you can retrieve afterward using `OCIXStreamInProcessedLWMGet()`. Call this function only when there is no LCR to send to the server and the client wants to know the progress of the attached inbound server.

This call returns `OCI_ERROR` if it is invoked from the callback functions of `OCIXStreamInLCRCallbackSend()`.

## 35.3.30 OCIXStreamInChunkSend()

Sends a chunk to the inbound server.

### Purpose

This function is valid during the execution of the `OCIXStreamInLCRSend()` call.

### Syntax

```
sword OCIXStreamInChunkSend (  OCISvcCtx   *svchp,
                               OCIError    *errhp,
                               oratext     *column_name,
                               ub2          column_name_len,
                               ub2          column_dty,
                               oraub8       column_flag,
                               ub2          column_csid,
                               ub4          chunk_bytes,
                               ub1         *chunk_data,
                               oraub8       flag,
                               ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**column_name (IN)**
Name of column associated with the given data. Column name must be canonicalized and must follow Oracle Database naming convention.

**column_name_len (IN)**
Length of column name.

**column_dty (IN)**
LCR chunk data type (must be `SQLT_CHR` or `SQLT_BIN`). See Table 35-5.

**column_flag (IN)**
Column flag. (See Comments for valid column flags.) Must specify `OCI_LCR_COLUMN_LAST_CHUNK` for the last chunk of each LOB or `LONG` or `XMLType` column.

**column_csid (IN)**
Column character set ID. This is required only if the `column_flag` has `OCI_LCR_COLUMN_XML_DATA` bit set.

**chunk_bytes (IN)**
Chunk data length in bytes.

**chunk_data (IN)**
Pointer to column data chunk. If the column is `NCLOB` or varying width `CLOB`, then the input chunk data must be in `AL16UTF16` format. The chunk data must be in the character set defined in Table 35-5.

**flag (IN)**
If `OCI_XSTREAM_MORE_ROW_DATA` (0x01) bit is set, then the current row change contains more data. You must clear this bit when sending the last chunk of the current LCR.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

You can call `OCIXStreamInChunkSend()` to send a chunk of large argument data for a procedure LCR to the XStream Inbound server.

The following LCR column flags can be combined using bitwise `OR` operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */

#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */

#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */

#define OCI_LCR_COLUMN_32K_DATA     /* col contains 32K data */

#define OCI_LCR_COLUMN_OBJECT_DATA  /* col contains object data in xml format */
```

In Streams, LOB, `LONG`, or `XMLType` column data is broken up into multiple chunks. For a row change containing columns of these data types, its associated LCR only contains data for the other column types. All LOB, `LONG` or `XMLType` columns are either represented in the LCR as `NULL` or not included in the LCR as defined in Table 35-4.

`OCILCRRowColumnInfoSet()` is provided to generate a list of scalar columns in an LCR. For LOB, `LONG`, and `XMLType` columns, `OCIXStreamInChunkSend()` is provided to set the value of each chunk in a column. For a large column, this function can be invoked consecutively multiple times with smaller chunks of data. The XStream inbound server can assemble these chunks and apply the accumulated change to the designated column.

The LCR of a row change must contain all the scalar columns that can uniquely identify a row at the apply site. Table 35-4 describes the required column list in each LCR for each DML operation.

**Table 35-4 Required Column List in the First LCR**

| Command Type of the First LCR of a Row Change | Columns Required in the First LCR |
|---|---|
| `INSERT` | The `NEW` column list must contain all non-`NULL` scalar columns. All LOB, `XMLType`, and `LONG` columns with chunk data must be included in this NEW column list. Each must have `NULL` value and `OCI_LCR_COLUMN_EMPTY_LOB` flag specified. |
| `UPDATE` | The `OLD` column list must contain the key columns.<br>The `NEW` column list must contain all updated scalar columns. All LOB, `XMLType`, and `LONG` columns with chunk data must be included in this NEW column list. Each must have `NULL` value and `OCI_LCR_COLUMN_EMPTY_LOB` flag specified. |
| `DELETE` | The `OLD` column list must contain the key columns. |
| `LOB_WRITE`, `LOB_TRIM`, `LOB_ERASE` | The `NEW` column list must contain the key columns and the modified LOB column. |

After constructing each LCR, you can call `OCIXStreamInLCRSend()` to send that LCR. Afterward, `OCIXStreamInChunkSend()` can be called repeatedly to send the chunk data for each LOB or `LONG` or `XMLType` column in that LCR. Sending the chunk value for different columns cannot be interleaved. If a column contains multiple chunks, then this function must be called consecutively using the same column name before proceeding to a new column. The ordering of the columns is irrelevant.

When invoking this function, you must pass `OCI_XSTREAM_MORE_ROW_DATA` as the flag argument if there is more data for the current LCR. When sending the last chunk of the current LCR, then this flag must be cleared to signal the end of the current LCR.

This function is valid only for `INSERT`, `UPDATE`, and `LOB_WRITE` operations. Multiple LOB, `LONG`, or `XMLType` columns can be specified for `INSERT` and `UPDATE`, while only one LOB column is allowed for `LOB_WRITE` operation.

The following is a sample client pseudocode snippet for non-callback mode (error checking is not included for simplicity):

```
main
{
   /* Attach to inbound server */
   OCIXStreamInAttach();

   /* Get the server's processed low position to determine
    * the position of the first LCR to generate.
    */
   OCIXStreamInProcessedLWMGet(&lwm);

   while (TRUE)
   {
      flag = 0;
      /* construct lcr */
      OCILCRHeaderSet(lcr);
      OCILCRRowColumnInfoSet(lcr);

      if (lcr has LOB | LONG | XMLType columns)
```

```
            set OCI_XSTREAM_MORE_ROW_DATA flag;

         status = OCIXStreamInLCRSend(lcr, flag);

         if (status == OCI_STILL_EXECUTING &&
                (OCI_XSTREAM_MORE_ROW_DATA flag set))
         {
            for each LOB/LONG/XMLType column in row change
            {
               for each chunk in column
               {
                  /* set col_name, col_flag, chunk data */
                  construct chunk;

                  if (last chunk of current column)
                      col_flag |= OCI_LCR_COLUMN_LAST_CHUNK;

                  if (last chunk of last column)
                     clear OCI_XSTREAM_MORE_ROW_DATA flag;

                  OCIXStreamInChunkSend(chunk, col_flag, flag);
               }
            }
         }
         else if (status == OCI_SUCCESS)
         {
            /* get lwm when SendLCR call ends successfully. */
            OCIXStreamInProcessedLWMGet(&lwm);
         }

         if (some terminating_condition)
           break;
      }

      OCIXStreamInDetach();
   }
```

## 35.3.31 OCIXStreamInCommit()

Commits the given transaction.

### Purpose

This function lets the client notify the inbound server about a transaction that has been executed by the client rather than by the server. So that if the same transaction is retransmitted during apply restart, it is ignored by the inbound server. A commit LCR must be supplied for the inbound server to extract the transaction ID and the position of the commit.

### Syntax

```
sword OCIXStreamInCommit ( OCISvcCtx *svchp,
                           OCIError  *errhp,
                           void      *lcrp,
                           ub4        mode );
```

#### Parameters

**svchp (IN/OUT)**
OCI service handle.

**errhp (IN/OUT)**
Error Handle to which errors should be reported.

**lcrp (IN)**
Pointer to the LCR to send. Must be a commit LCR.

**mode (IN)**
Mode flags. Not used currently; used for future extension.

#### Comments

The position of the input LCR must be higher than `DBA_XSTREAM_INBOUND_PROGRESS.APPLIED_HIGH_POSITION`, and the LCR's source database must match `DBA_APPLY_PROGRESS.SOURCE_DATABASE` of the attached inbound server.

If there is any pre-commit handler defined, it is executed when this commit LCR is executed.

Assume a sample use case in which a situation where the inbound server does not support certain data types, but the client can do the work directly. The client performs the transaction changes directly to the database and then invokes the `OCIXStreamInCommit()` to commit the transaction by way of the inbound server. Note that the client should not directly commit the transaction itself. Rather, the transaction changes are committed with this command (`OCIXStreamInCommit()`) so that the transaction is atomic. Thus, if the inbound server becomes disabled during the client transaction, then the entire transaction is correctly rolled back.

## 35.3.32 OCIXStreamInSessionSet()

Sets session attributes for XStream In functions.

#### Purpose

Sets session attributes for XStream In functions.

#### Syntax

```
sword OCIXStreamInSessionSet(OCISvcCtx   *svchp,
                             OCIError    *errhp,
                             oratext     *attribute_name,
                             ub2          attribute_name_len,
                             void        *attribute_value,
                             ub2          attribute_value_len,
                             ub2          attribute_dty,
                             ub4          mode);
```

#### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**attribute_name (IN)**
An attribute name.
Valid values for `attribute_name` are:

```
#define  OCIXSTREAM_ATTR_ATTACH_TIMEOUT "ATTACH_TIMEOUT_SECS"
#define  OCIXSTREAM_ATTR_MAX_ATTACH_RETRIES "MAX_ATTACH_RETRIES"
```

**attribute_name_len (IN)**
An attribute name length.
The maximum value for `attribute_name_len` is `OCIXSTREAM_SESSION_SET_MAX_PARAM_LEN`:

```
#define  OCIXSTREAM_SESSION_SET_MAX_PARAM_LEN   128
```

**attribute_value (IN)**
The attribute value.

**attribute_value_len (IN)**
The attribute value length.
The maximum value for `attribute_value_len` is 128.

**attribute_dty (IN)**
Pointer to an array of attribute data types. The only valid value for `attribute_dty` is `DTYUB2`. An error is returned if you try a data type other than `DTYUB2` for `ATTACH_TIMEOUT_SECS` and `MAX_ATTACH_RETRIES`.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

You must invoke `OCIXStreamInSessionSet()` before calling `OCIXStreamInAttach()`.

### Returns

`OCI_SUCCESS` if successful, otherwise `OCI_ERROR`.

## 35.3.33 OCIXStreamOutAttach()

Attaches to an XStream outbound server.

### Purpose

The client application must connect to the database using a dedicated connection.

### Syntax

```
sword OCIXStreamOutAttach ( OCISvcCtx   *svchp,
                            OCIError    *errhp,
                            oratext     *server_name,
                            ub2          server_name_len,
                            ub1         *last_position,
                            ub2          last_position_len,
                            ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**server_name (IN)**
XStream outbound server name.

**server_name_len (IN)**
Length of XStream outbound server name.

**last_position (IN)**
Position to the last received LCR. Can be `NULL`.

**last_position_len (IN)**
Length of `last_position`.

**mode (IN)**
`OCIXSTREAM_OUT_ATTACH_APP_FREE_LCR` - If this mode is specified, then the application is in charge of freeing the LCRs from the outbound server.
`OCIXSTREAM_OUT_ATTACH_APP_CONTAINER` - If this mode is specified, then the application is capturing application container statements (ALTER PLUGGABLE DATABASE APPLICATIONS LCRs). The XStream application should set this mode option if it is intending to replicate ALTER PLUGGABLE DATABASE APPLICATIONS LCRs to other application containers.
`OCIXSTREAM_OUT_ATTACH_EXTENDED_TXID` - If this mode is specified, then the application is making a request for extended transaction ID format. The typical transaction ID format is `xidusn.xidslt.xidsqn`. The extended transaction ID format is `pdbuid.xidusn.xidslt.xidsqn`. Use this mode if the XStream Out server is configured to capture changes from multiple containers and the source database has local undo mode enabled.

### Comments

The `OCIEnv` environment handle must be created with `OCI_OBJECT` mode, and the service context must be in a connected state to issue this function. This function does not support nonblocking mode. It returns either the `OCI_SUCCESS` or `OCI_ERROR` status code.

The name of the outbound server must be provided because multiple outbound servers can be configured in one Oracle Database instance. This function returns `OCI_ERROR` if it encounters any error while attaching to the outbound server. Only one client can attach to an XStream outbound server at any time. An error is returned if multiple clients attempt to attach to the same outbound server or if the same client attempts to attach to multiple outbound servers using the same service handle.

The `last_position` parameter is used to establish the starting point of the stream. This call returns `OCI_ERROR` if the specified position is non-`NULL` and less than the server's processed low position (see `OCIXStreamOutProcessedLWMSet()`); otherwise, LCRs with positions greater than the specified `last_position` are sent to the user.

If the `last_position` is `NULL`, then the stream starts from the processed low position maintained in the server.

**Related Topics**

- OCIXStreamOutProcessedLWMSet()
  Updates the local copy of the processed low position.

## 35.3.34 OCIXStreamOutDetach()

Detaches from the outbound server.

### Purpose

Detaches from the outbound server.

### Syntax

```
sword OCIXStreamOutDetach ( OCISvcCtx   *svchp,
                            OCIError    *errhp,
                            ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle that you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

This function sends the current local processed low position to the server before detaching from the outbound server. The outbound server automatically restarts after this call. This function returns `OCI_ERROR` if it is invoked while a `OCIXStreamOutReceive()` call is in progress.

## 35.3.35 OCIXStreamOutLCRReceive()

Receives an LCR, including a procedure LCR, from an outbound stream.

### Purpose

If an LCR is available, then this function immediately returns that LCR. The duration of each LCR is limited to the interval between two successive `OCIXStreamOutLCRReceive()` calls. When there is no LCR available in the stream, this call returns a `NULL` LCR after an idle timeout.

### Syntax

```
sword OCIXStreamOutLCRReceive ( OCISvcCtx     *svchp,
                                OCIError      *errhp,
                                void         **lcrp,
                                ub1           *lcrtype,
                                       oraub8        *flag,
                                       ub1           *fetch_low_position,
                                       ub2           *fetch_low_position_len,
                                       ub4            mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**lcrp (OUT)**
Pointer to the LCR received from the stream; can include a procedure LCR. If there is an available LCR, then this LCR is returned with status code `OCI_STILL_EXECUTING`. When the call ends, a `NULL` LCR is returned with status code `OCI_SUCCESS`.

**lcrtype (OUT)**
This value is valid only when `lcrp` is not `NULL`.

**flag (OUT)**
Return flag. If bit `OCI_XSTREAM_MORE_ROW_DATA` (0x01) is set, then this LCR has more data. You must use `OCIXStreamOutReceiveChunk()` function to get the remaining data.

**fetch_low_position (OUT)**
XStream outbound server's fetch low position. This value is returned only when the return code is `OCI_SUCCESS`. Optional. If non-`NULL`, then you must preallocate `OCI_LCR_MAX_POSITION_LEN` bytes for the return value.

**fetch_low_position_len (OUT)**
Length of `fetch_low_position`.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

To avoid a network round-trip for every `OCIXStreamOutLCRReceive()` call, the connection is tied to this call and allows the server to fill up the network buffer with LCRs so subsequent calls can quickly receive the LCRs from the network. The server ends each call at the transaction boundary after an ACK interval elapses since the call began. When there is no LCR in the stream, the server ends the call after the idle timeout elapses.

Return codes:

- `OCI_STILL_EXECUTING` means that the current call is still in progress. The connection associated with the specified service context handle is still tied to this call for streaming the LCRs from the server. An error is returned if you attempt to use the same connection to execute any OCI calls that require database round-trip, for example, `OCIStmtExecute()`, `OCIStmtFetch()`, `OCILobRead()`, and so on. `OCILCR*` calls do not require round-trips; thus, they are valid while the call is in progress.

- `OCI_SUCCESS` means that the current call is completed. You are free to execute `OCIStmt*`, `OCILob*`, and so on from the same service context.
- `OCI_ERROR` means the current call encounters some errors. Use `OCIErrorGet()` to obtain information about the error.

This call always returns a `NULL` LCR when the return code is `OCI_SUCCESS`. In addition, it returns the fetch low position to denote that the outbound server has received all transactions with commit position lower than or equal to this value.

> **See Also:**
>
> - Server Handle Attributes
> - OCIXStreamOutChunkReceive() for non-callback pseudocode in the Comments section

## 35.3.36 OCIXStreamOutLCRCallbackReceive()

Used to get the LCR stream from the outbound server using callbacks.

**Purpose**

You must supply a callback procedure to be invoked for each LCR received. If some LCRs in the stream may contain `LOB` or `LONG` or `XMLType` columns, then a second callback must be supplied to process each chunk (see `OCIXStreamOutChunkReceive()`).

**Syntax**

```
sword OCIXStreamOutLCRCallbackReceive (
           OCISvcCtx                        *svchp,
           OCIError                         *errhp,
           OCICallbackXStreamOutLCRProcess   processlcr_cb,
           OCICallbackXStreamOutChunkProcess processchunk_cb,
           void                             *usrctxp,
           ub1                              *fetch_low_position,
           ub2                              *fetch_low_position_len,
           ub4                               mode );
```

**Parameters**

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**processlcr_cb (IN)**
Callback function to process each LCR received by the client. Cannot be `NULL`.

**processchunk_cb (IN)**
Callback function to process each chunk in the received LCR. Can be `NULL` if you do not expect to receive any LCRs with additional chunk data.

**usrctxp (IN)**
User context to pass to both callback procedures.

**fetch_low_position (OUT)**
XStream outbound server's fetch low position (see `OCIXStreamOutLCRReceive()`). Optional.

**fetch_low_position_len (OUT)**
Length of `fetch_low_position`.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

Return code: `OCI_SUCCESS` or `OCI_ERROR`.

The `processlcr_cb` argument must be of type `OCICallbackXStreamOutLCRProcess`:

```
typedef sb4  (*OCICallbackXStreamOutLCRProcess)
               (void  *usrctxp, void *lcrp, ub1 lcrtyp, oraub8 flag);
```

Parameters of `OCICallbackXStreamOutLCRProcess()`:

**usrctxp (IN/OUT)**
Pointer to the user context.

**lcrp (IN)**
Pointer to the LCR just received.

**lcrtyp (IN)**
LCR type (`OCI_LCR_XROW` or `OCI_LCR_XDDL`).

**flag (IN)**
If `OCI_XSTREAM_MORE_ROW_DATA` is set, then the current LCR has more chunk data.

The input parameters of the `processlcr_cb()` procedure are the user context, the LCR just received, its type, and a flag to indicate whether the LCR contains more data. If there is an LCR available, then this callback is invoked immediately. If there is no LCR in the stream, after an idle timeout, then this call ends with `OCI_SUCCESS` return code.

You can invoke the callback `OCICallbackXStreamOutLCRProcess` to process procedure LCRs as well.

The valid return codes from the `OCICallbackXStreamOutLCRProcess()` callback function are `OCI_CONTINUE` or `OCI_SUCCESS`. This callback function must return `OCI_CONTINUE` to continue processing the `OCIXStreamOutLCRCallbackReceive()` call. Any return code other than `OCI_CONTINUE` signals that the client wants to terminate `OCIXStreamOutLCRCallbackReceive()` immediately.

**See Also:**

Server Handle Attributes

The `processchunk_cb` argument must be of type `OCICallbackXStreamOutChunkProcess`:

```
typedef sb4  (*OCICallbackXStreamOutChunkProcess)
(void        *usrctxp,
oratext      *column_name,
ub2          column_name_len,
ub2          column_dty,
oraub8       column_flag,
ub2          column_csid,
ub4          chunk_bytes,
ub1          *chunk_data,
oraub8       flag );
```

Parameters of `OCICallbackXStreamOutChunkProcess()`:

**usrctxp (IN/OUT)**
Pointer to the user context.

**column_name (IN)**
Column name of the current chunk.

**column_name_len (IN)**
Length of the column name.

**column_name_dty (IN)**
Chunk data type (`SQLT_CHR` or `SQLT_BIN`).

**column_flag (IN)**
See Comments in `OCIXStreamInChunkSend()`.

**column_csid (IN)**
Column character set ID. Relevant only if the column is an `XMLType` column (that is, `column_flag` has the `OCI_LCR_COLUMN_XML_DATA` bit set).

**chunk_bytes (IN)**
Chunk data length in bytes.

**chunk_data (IN)**
Chunk data pointer.

**flag (IN)**
If `OCI_XSTREAM_MORE_ROW_DATA` is set, then the current LCR has more chunk data.

The input parameters of the `processchunk_cb()` procedure are the user context, the information about the chunk, and a flag. When the `flag` argument has the `OCI_XSTREAM_MORE_ROW_DATA` (0x01) bit set, then there is more data for the current LCR.

The valid return codes from the `OCICallbackXStreamOutChunkProcess()` callback function are `OCI_CONTINUE` or `OCI_SUCCESS`. This callback function must return

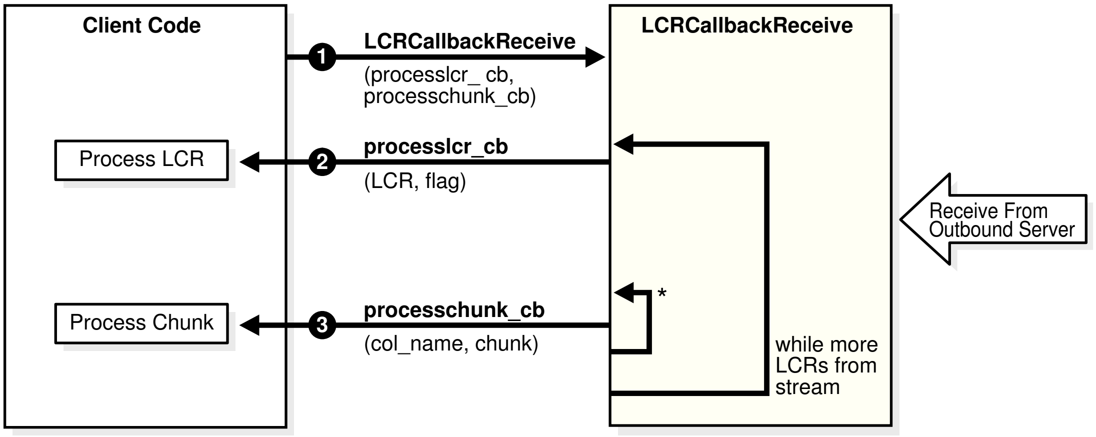
`OCI_CONTINUE` to continue processing the `OCIXStreamOutLCRCallbackReceive()` call. Any return code other than `OCI_CONTINUE` signals that the client wants to terminate `OCIXStreamOutLCRCallbackReceive()` immediately.

OCI calls are provided to access each field in the LCR. If the LCR contains only scalar column(s), then the duration of that LCR is limited only to the `processlcr_cb()` procedure. If the LCR contains some chunk data, then the duration of the LCR is extended until all the chunks have been processed. If you want to access the LCR data at a later time, then a copy of the LCR must be made before it is freed.

As for `OCIXStreamOutLCRReceive()`, the server ends each call at the transaction boundary after each ACK interval since the call began, or after each idle timeout. The default ACK interval is 30 seconds, and the default idle timeout is one second. See "Server Handle Attributes" to tune these values. This function also returns the fetch low position when the call ends.

Figure 35-2 shows the execution flow of the `OCIXStreamOutLCRCallbackReceive()` function.

**Figure 35-2 Execution Flow of the OCIXStreamOutLCRCallbackReceive() Function**


Client Code
Process LCR
Process Chunk
1
LCRCallbackReceive
(processlcr_ cb, processchunk_cb)
2
processlcr_cb
(LCR, flag)
3
processchunk_cb
(col_name, chunk)
LCRCallbackReceive
*
while more LCRs from stream
Receive From Outbound Server

* While `OCI_XSTREAM_MORE_ROW_DATA` is set.

Description of Figure 35-2:

- At 1, the client invokes `OCIXStreamOutLCRCallbackReceive()` providing two callbacks. This function initiates an LCR outbound stream from the server.
- At 2, this function receives an LCR from the stream and invokes `processlcr_cb()` procedure with the LCR just received. It passes `OCI_XSTREAM_MORE_ROW_DATA` flag to `processlcr_cb()` if the current LCR has additional data.
- If the current LCR has no additional chunk, then this function repeats 2 for the next LCR in the stream.
- At 3, if the current LCR contains additional chunk data, then this function invokes `processchunk_cb()` for each chunk received with the `OCI_XSTREAM_MORE_ROW_DATA` flag. This flag is cleared when the callback is invoked on the last chunk of the current LCR.

- If there is more LCR in the stream, then it loops back to 2. This process continues until the end of the current call, or when there is no LCR in the stream for one second, or if a callback function returns any value other than `OCI_CONTINUE`.

Here is sample pseudocode for callback mode:

```
main
{
   /* Attach to outbound server specifying last position */
   OCIXStreamOutAttach(last_pos);

   /* Update the local processed low position */
   OCIXStreamOutProcessedLWMSet(lwm);

   while (TRUE)
   {
      OCIXStreamOutLCRCallbackReceive(processlcr_cb,
                                      processchunk_cb, fwm);

      /* Use fetch low position(fwm)
       * to update processed lwm if applied.
       */

      /* Update the local lwm so it is sent to
       * server during next call.
       */
      OCIXStreamOutProcessedLWMSet(lwm);
      if (some terminating_condition)
         break;
   }
   OCIXStreamOutDetach();
}

processlcr_cb (IN lcr, IN flag)
{
   /* Process the LCR just received */
   OCILCRHeaderGet(lcr);
   OCILCRRowColumnInfoGet(lcr);

   if (lcr is LOB_WRITE | LOB_TRIM | LOB_ERASE)
      OCILCRLobInfoGet(lcr);

   if (OCI_XSTREAM_MORE_ROW_DATA flag set)
      prepare_for_chunk_data;
   else
      process_end_of_row;
}

processchunk_cb (IN chunk, IN flag)
{
   process_chunk;

   if (OCI_XSTREAM_MORE_ROW_DATA flag not set)
      process_end_of_row;
}
```

**Related Topics**

- OCIXStreamOutChunkReceive()
  Retrieves data of each LOB or LONG or XMLType column one chunk at a time.

- OCIXStreamOutLCRReceive()
  Receives an LCR, including a procedure LCR, from an outbound stream.
- OCIXStreamInChunkSend()
  Sends a chunk to the inbound server.

## 35.3.37 OCIXStreamOutProcessedLWMSet()

Updates the local copy of the processed low position.

### Purpose

This function can be called anytime between `OCIXStreamOutAttach()` and `OCIXStreamOutDetach()` calls. Clients using the callback mechanism to stream LCRs from the server (see `OCIXStreamOutLCRCallbackReceive()`), can invoke this function while in the callback procedures.

### Syntax

```
sword OCIXStreamOutProcessedLWMSet ( OCISvcCtx  *svchp,
                                     OCIError   *errhp,
                                     ub1        *processed_low_position,
                                     ub2         processed_low_position_len,
                                     ub4         mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**processed_low_position (IN)**
The processed low position maintained at the client.

**processed_low_position_len (IN)**
Length of `processed_low_position`.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

The processed low position denotes that all LCRs at or below it have been processed. After successfully attaching to an XStream outbound server, a local copy of the processed low position is maintained at the client. Periodically, this position is sent to the server so that archived redo log files containing already processed transactions can be purged.

Return code: `OCI_SUCCESS` or `OCI_ERROR`.

Clients using `XStreamOut` functions must keep track of the processed low position based on what they have processed and call this function whenever their processed low position has changed. This is done so that a more current value is sent to the server during the next update, which occurs at the beginning of the `OCIXStreamOutLCRCallbackReceive()` and `OCIXStreamDetach()` calls. For an

`OCIXStreamOutLCRReceive()` call, the processed low position is sent to the server when it initiates a request to start the outbound stream. It is not sent while the stream is in progress.

You can query the `DBA_XSTREAM_OUTBOUND_PROGRESS` view to confirm that the processed low position has been saved in the server.

**Related Topics**

- OCIXStreamOutLCRCallbackReceive()
  Used to get the LCR stream from the outbound server using callbacks.

## 35.3.38 OCIXStreamOutChunkReceive()

Retrieves data of each LOB or LONG or XMLType column one chunk at a time.

### Purpose

Allows the client to retrieve the data of each LOB or `LONG` or `XMLType` column one chunk at a time.

### Syntax

```
sword OCIXStreamOutChunkReceive ( OCISvcCtx   *svchp,
                                  OCIError    *errhp,
                                  oratext    **column_name,
                                  ub2         *column_name_len,
                                  ub2         *column_dty,
                                  oraub8      *column_flag,
                                  ub2         *column_csid,
                                  ub4         *chunk_bytes,
                                  ub1        **chunk_data,
                                  oraub8      *flag,
                                  ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**column_name (OUT)**
Name of the column that has data.

**column_name_len (OUT)**
Length of the column name.

**column_dty (OUT)**
Column chunk data type (either `SQLT_CHR` or `SQLT_BIN`).

**column_flag (OUT)**
Column flag. See Comments for valid flags.

**column_csid (OUT)**
Column character set ID. This is returned only for `XMLType` column, that is, `column_flag` has `OCI_LCR_COLUMN_XML_DATA` bit set.

**chunk_bytes (OUT)**
Number of bytes in the returned chunk.

**chunk_data (OUT)**
Pointer to the chunk data in the LCR. The client must not deallocate this buffer since the LCR and its contents are maintained by this function.

**flag (OUT)**
If `OCI_XSTREAM_MORE_ROW_DATA` (0x01) is set, then the current LCR has more chunks coming.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

In Streams, LOB, `LONG`, or `XMLType` column data is broken up into multiple LCRs based on how they are stored in the online redo log files. Thus, for a row change containing these columns multiple LCRs may be constructed. The first LCR of a row change contains the column data for all the scalar columns. All LOB or `LONG` or `XMLType` columns in the first LCR are set to `NULL` because their data are sent in subsequent LCRs for that row change. These column data are stored in the LCR as either `RAW` (`SQLT_BIN`) or `VARCHAR2` (`SQLT_CHR`) chunks as shown in Table 35-5.

**Table 35-5 Storage of LOB or LONG Data in the LCR**

| Source Column Data Type | Streams LCR Data Type | Streams LCR Character Set |
|---|---|---|
| `BLOB` | `RAW` | N/A |
| Fixed-width `CLOB` | `VARCHAR2` | Client Character Set |
| Varying-width `CLOB` | `RAW` | AL16UTF16 |
| `NCLOB` | `RAW` | AL16UTF16 |
| `XMLType` | `RAW` | Column `csid` obtained from the chunk |

In Streams, LOB, `LONG`, or `XMLType` column data is broken up into multiple chunks based on how they are stored in the online redo log files. For a row change containing columns of these data types, its associated LCR only contains data for the other scalar columns. All LOB, `LONG`, or `XMLType` columns are either represented in the LCR as `NULL` or not included in the LCR. The actual data for these columns are sent following each LCR as `RAW` (`SQLT_BIN`) or `VARCHAR2` (`SQLT_CHR`) chunks as shown in Table 35-5.

The following LCR column flags can be combined using the bitwise `OR` operator.

```
#define OCI_LCR_COLUMN_LOB_DATA     /* column contains LOB data */
#define OCI_LCR_COLUMN_LONG_DATA    /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB    /* column has an empty LOB  */
#define OCI_LCR_COLUMN_LAST_CHUNK   /* last chunk of current column */
#define OCI_LCR_COLUMN_AL16UTF16    /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB        /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA     /* column contains xml data */
```

```
#define OCI_LCR_COLUMN_XML_DIFF     /* column contains xmldiff data */

#define OCI_LCR_COLUMN_ENCRYPTED    /* column is encrypted */
#define OCI_LCR_COLUMN_UPDATED      /* col is updated */

#define OCI_LCR_COLUMN_32K_DATA     /* col contains 32K data */

#define OCI_LCR_COLUMN_OBJECT_DATA  /* col contains object data in xml format */
```

Return code: `OCI_ERROR` or `OCI_SUCCESS`.

This call returns a `NULL` column name and `NULL` chunk data if it is invoked when the current LCR does not contain the LOB, `LONG`, or `XMLType` columns. This function is valid only when an `OCIXStreamOutLCRReceive()` call is in progress. An error is returned if it is called during other times.

If the return flag from `OCIXStreamOutLCRReceive()` has `OCI_XSTREAM_MORE_ROW_DATA` bit set, then you must iteratively call `OCIXStreamOutChunkReceive()` to retrieve all the chunks belonging to that row change before getting the next row change (that is, before making the next `OCIXStreamOutLCRReceive()` call); otherwise, an error is returned.

Here is sample pseudocode for non-callback mode:

```
main
{
   /* Attach to outbound server specifying last position */
   OCIXStreamOutAttach(last_pos);

   /* Update the local processed low position */
   OCIXStreamOutProcessedLWMSet(lwm);

   while (TRUE)
   {
      status = OCIXStreamOutLCRReceive(lcr, flag, fwm);

      if (status == OCI_STILL_EXECUTING)
      {
         /* Process LCR just received */
         OCILCRHeaderGet(lcr);
         OCILCRRowColumnInfoGet(lcr);

         while (OCI_XSTREAM_MORE_ROW_DATA flag set)
         {
            OCIXStreamReceiveChunk(chunk, flag, );

            process_chunk;
         }
         process_end_of_row;
      }
      else if (status == OCI_SUCCESS)
      {
         /* Use fetch low position(fwm)
          * to update processed lwm if applied.
          */

         /* Update the local lwm so it is sent to
          * server during next call.
          */
         OCIXStreamOutProcessedLWMSet(lwm);
```

```
          if (some terminating_condition)
            break;
       }
   }
   OCIXStreamOutDetach();
}
```

## 35.3.39 OCIXStreamOutGetNextChunk()

While in the `ProcessLCRStream()` callback function, retrieves the next chunk in the current LCR until all chunks are retrieved.

### Purpose

While in the `ProcessLCRStream()` callback function, the client must repeatedly call `OCIXStreamOutGetNextChunk()` to retrieve all the chunks in the current LCR (that is, until either `OCI_ERROR` or `OCI_SUCCESS` is returned). If this is not done, an error will be raised when the callback function ends.

### Syntax

```
sword OCIXStreamOutGetNextChunk ( OCISvcCtx   *svchp,
                                  OCIError    *errhp,
                                  oratext    **column_name,
                                  ub2         *column_name_len,
                                  ub2         *column_dty,
                                  oraub8      *column_flag,
                                  ub4         *rtn_bytes,
                                  ub1        **bufp,
                                  ub4         *op_offset,
                                  ub4         *op_size,
                                  ub4          mode );
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN/OUT)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

**column_name (OUT)**
Name of LOB column which the data belongs to.

**column_name_len (OUT)**
Length of column name.

**column_dty (OUT)**
Column data type (either `SQLT_CHR` or `SQLT_BIN`).

**column_flag (OUT)**
Column flag. Valid values are listed in Comments.

**rtn_bytes (OUT)**
Number of bytes in returned buffer.

**bufp (OUT)**
Pointer to the LOB chunk data in the LCR. The client should not de-allocate this buffer since the LCR and its contents are maintained by the function.

**op_offset (OUT)**
LOB operation offset. Optional. Data is not returned if the argument is `NULL`. Relevant only for `LOB_WRITE` and `LOB_TRIM` operation; otherwise, a zero is returned.

**op_size (OUT)**
LOB operation size. Optional. Data not returned if the argument is `NULL`. Relevant only for `LOB_TRIM` and `LOB_ERASE` operation; otherwise, a zero is returned.

**mode (IN)**
Specify `OCI_DEFAULT` for now.

### Comments

The `OCIEnv` environment handle must be created with `OCI_OBJECT` mode, and the service context must be in connected state to issue this function.

In streams, LOB or `LONG` column data is broken up into multiple LCRs based on how they are stored in the redo logs. Thus, for a row change containing Re `LONG` columns, multiple LCRs can be constructed. The first LCR of a row change contains the column data for all the scalar columns. All LOB or `LONG` columns in the first LCR are set to `NULL` since their data are sent in subsequent LCRs for that row change. `LONG` column data is implicitly converted to LOB before the data is sent to the client. LOB or `LONG` data is stored in the LCR as `RAW` (`SQLT_BIN`) or `VARCHAR2` (`SQLT_CHR`) as shown in Table 35-5:

The `OCIXStreamOutProcessLCRStream()` function allows the client to retrieve the data of each LOB or LONG column one chunk at a time. All the chunks belonging to one column are returned in a contiguous order. Chunk data from one column is returned entirely before switching to the next column. There is no fixed ordering on how the columns are returned. The client should not rely on the ordering of LOB columns but rely on the returned column names.

Invoking `OCIXStreamOutGetNextChunk()` when the current LCR contains only scalar columns results in a `NULL` column name, and no data is returned.

The returned column flag provides more information on each chunk data. For example, the flag `OCI_LCR_COLUMN_LAST_CHUNK` is always returned on the last chunk of each column:

The following flags can specify a column property. Multiples of these flags can be combined using the bitwise `OR` operator.

```
#define OCI_LCR_COLUMN_LOB_DATA /* column contains lob data */
#define OCI_LCR_COLUMN_LONG_DATA /* column contains long data */
#define OCI_LCR_COLUMN_EMPTY_LOB /* column has an empty lob */
#define OCI_LCR_COLUMN_LAST_CHUNK /* last chunk of column */
#define OCI_LCR_COLUMN_AL16UTF16 /* column is in AL16UTF16 fmt */
#define OCI_LCR_COLUMN_NCLOB /* column has NCLOB data */
#define OCI_LCR_COLUMN_XML_DATA /* column contains xml data */
#define OCI_LCR_COLUMN_XML_DIFF /* column contains xmldiff */
#define OCI_LCR_COLUMN_ENCRYPTED /* column is encrypted */
```

During an `OCIXStreamOutProcessLCRStream()` call, the client provides a callback function that will be invoked each time an LCR is retrieved. For an LCR containing

LOB or `LONG` columns, the first LCR of a row change is passed to this callback function. From within this callback function, the client can repeatedly call `OCIXStreamOutGetNextChunk()` to retrieve all the chunks for each LOB or `LONG` column. When this function returns `OCI_SUCCESS`, it means the client has consumed all the data for the current row change.

While in the `ProcessLCRStream` callback function the client must repeatedly call `OCIXStreamOutGetNextChunk()` to retrieve all the chunks in the current LCR (that is, until either `OCI_ERROR` or `OCI_SUCCESS` is returned). If this is not done, an error is raised when the callback function ends.

This function is valid only during the duration that the `ProcessLCRStream` callback function is invoked. An error is raised if it is called during other times.

This function returns the following values:

- `OCI_SUCCESS` means that all the data for the current LCR has been retrieved. You can check the returned column and buffer information to access the returned column data.
- `OCI_NEED_DATA` means there is more data in the stream for the current LCR. You can check the returned column and buffer information to access the returned column data.
- `OCI_ERROR` means there is an error. You can use `OCIErrorGet()` to get the error code.

Following is a sample client code snippet (error checking is not included for simplicity):

```
main()
{
...
OCIServerAttach(...); /* Connect to Oracle db */
/* Attach to the outbound server named ‘TT1' */
OCIXStreamOutAttach(...,“TT1", 3, hwm, hwm_len, (OCIDate *)0, OCI_DEFAULT);
/*-----------------------------------------------------
Initialize the LWM by querying the client's system tables
or set to zero and let the outbound server determines the LWM.
------------------------------------------------------*/
Init_lwm(&lwm);
/*------------------------------------------------------
ProcessLCR_CB is a callback function to process each LCR (see below).
usrctxp is the user context to be passed to the callback function.
------------------------------------------------------*/
while (OCIXStreamOutProcessLCRStream(..., lwm, lwm_len,(OCIDate *)0,
       ProcessLCR_CB, usrctxp, 0, OCI_DEFAULT) == OCI_SUCCESS)
{
/* Client maintains LWM. */
Maintain_lwm(&lwm);
...
}
/* detach from the outbound server */
OCIXStreamOutDetach(..., lwm, lwm_len, (OCIDate *)0, OCI_DEFAULT);
}
/* Sample callback routine to OCIXStreamOutProcessLCRStream() */
sb4 ProcessLCR_CB (void *ctxp, void *lcrp, ub1 lcrtyp)
{
/* Get header information for current LCR */
status = OCILCRGetHeader(lcrp,...);
if (status != OCI_SUCCESS)
    handle_error();
if (lcrtype == OCI_LCR_XROW)
  {
   if (OCILcrRowGetColumnInfo(lcrp, ...) != OCI_SUCCESS)
     handle_error();
   process_scalar_columns(...);
   do
   {
     status = OCIXStreamOutGetNextChunk(..., &colname, &colnamelen,
                                        &coldty, &flag, &rtn_bytes,
                                        bufp, ...);
     if (status == OCI_ERROR)
          handle_error();
     process_lob_chunk(...);
   } while (status == OCI_NEED_DATA);
       if (status == OCI_ERROR)
         handle_error();
  }
return OCI_CONTINUE;
}
```

## 35.3.40 OCIXStreamOutSessionSet()

Sets session attributes for XStream Out functions.

### Purpose

Sets session attributes for XStream Out functions.

### Syntax

```
sword OCIXStreamOutSessionSet(OCISvcCtx   *svchp,
                              OCIError    *errhp,
                              oratext     *attribute_name,
                              ub2          attribute_name_len,
                              void        *attribute_value,
                              ub2          attribute_value_len,
                              ub2          attribute_dty,
                              ub4          mode);
```

### Parameters

**svchp (IN)**
Service handle context.

**errhp (IN)**
An error handle you can pass to `OCIErrorGet()` for diagnostic information in case of an error.

**attribute_name (IN)**
An attribute name.
Valid values for `attribute_name` are:

```
#define  OCIXSTREAM_ATTR_ATTACH_TIMEOUT "ATTACH_TIMEOUT_SECS"
#define  OCIXSTREAM_ATTR_MAX_ATTACH_RETRIES "MAX_ATTACH_RETRIES"
```

**attribute_name_len (IN)**
An attribute name length.

The maximum value for `attribute_name_len` is `OCIXSTREAM_SESSION_SET_MAX_PARAM_LEN`:

```
#define  OCIXSTREAM_SESSION_SET_MAX_PARAM_LEN   128
```

**attribute_value (IN)**
The attribute value.

**attribute_value_len (IN)**
The attribute value length.
The maximum value for `attribute_value_len` is 128.

**attribute_dty (IN)**
Pointer to an array of attribute data types. The only valid value for `attribute_dty` is `DTYUB2`. An error is returned if you try a data type other than `DTYUB2` for `ATTACH_TIMEOUT_SECS` and `MAX_ATTACH_RETRIES`.

**mode (IN)**
Specify `OCI_DEFAULT`.

### Comments

You must invoke `OCIXStreamOutSessionSet()` before calling `OCIXStreamOutAttach()`.

### Returns

`OCI_SUCCESS` if successful, otherwise `OCI_ERROR`.

# 36

# OCI SODA Functions

This chapter describes the OCI SODA functions.

This chapter contains these topics:

- Introduction to OCI SODA Functions
- OCI SODA Functions

## 36.1 Introduction to OCI SODA Functions

The Simple Oracle Document (SODA) for C API is part of Oracle Call Interface (OCI).

SODA for C allows OCI programmers to store and query Javascript Object Notation (JSON) data and access JSON functionality in the database with abstractions: databases, collections, and documents. A database contains collections and collections contain documents. The API provides create, read, update, and delete (CRUD) operations on the document collections.

### Conventions for OCI Functions

See Conventions for OCI Functions for the conventions used in describing each function. The entries for each function may also contain the following information: Returns.

**See Also:**

- *Oracle Database SODA for C Developer's Guide*
- *Oracle Database JSON Developer's Guide*

## 36.2 OCI SODA Functions

The following table lists the OCI SODA functions that are described in this chapter.

**Table 36-1 OCI SODA Functions**

| Function | Purpose |
|---|---|
| `OCISodaBulkInsert()` | Inserts a document array into a collection. |
| `OCISodaBulkInsertAndGet()` | Inserts a document array into a collection and gets the result document array back. |
| `OCISodaBulkInsertAndGetWithCtnt()` | Inserts a document array into a collection and gets the result document array back. The input documents are represented as an array of content strings and an array of keys. |

**Table 36-1 (Cont.) OCI SODA Functions**

| Function | Purpose |
|---|---|
| `OCISodaBulkInsertWithCtnt()` | Inserts an array of documents into a collection. The documents are represented as an array of content strings and an array of keys. |
| `OCISodaCollCreate()` | Creates a new document collection with default metadata settings. |
| `OCISodaCollCreateWithMetadata()` | Creates a new document collection with metadata. |
| `OCISodaCollDrop()` | Drops a collection from the database. |
| `OCISodaCollGetNext()` | Gets the next collection from the collection cursor. |
| `OCISodaCollList()` | Lists the collections in the schema associated with the supplied service context handle parameter. |
| `OCISodaCollOpen()` | Opens the collection. |
| `OCISodaDataGuideGet()` | Gets JSON data guide information. |
| `OCISodaDocCount()` | Counts the number of documents from a collection. |
| `OCISodaDocCountWithFilter()` | Counts the number of documents in a collection based on the query-by-example filter document. |
| `OCISodaDocCreate()` | Creates a document. |
| `OCISodaDocCreateWithKey()` | Creates a document with document key and content. |
| `OCISodaDocCreateWithKeyAndMType()` | Creates a document with document key, content, and media type. |
| `OCISodaDocGetNext()` | Gets the next document in the collection. |
| `OCISodaFind()` | Find documents in a collection. |
| `OCISodaFindOne()` | Finds a single document in a collection. |
| `OCISodaFindOneWithKey()` | Finds a single document in a collection given a key. |
| `OCISodaIndexCreate()` | Creates an index with an index specification. |
| `OCISodaIndexDrop()` | Drops an index. |
| `OCISodaInsert()` | Inserts a document into a collection. |
| `OCISodaInsertAndGet()` | Inserts a document into a collection and returns a result document. |
| `OCISodaInsertAndGetWithCtnt()` | Inserts a document into a collection and returns a result document. The input document is represented as a content string and a key |
| `OCISodaInsertWithCtnt()` | Inserts a document into a collection. The document is represented as a content string and a key |
| `OCISodaRemove()` | Removes matching documents from a collection. |

**Table 36-1 (Cont.) OCI SODA Functions**

| Function | Purpose |
|---|---|
| `OCISodaRemoveOneWithKey()` | Removes a document from a collection given a key. |
| `OCISodaReplOne()` | Replaces a document in a collection. |
| `OCISodaReplOneAndGet()` | Replaces a document in a collection. The input document is represented only by the content string. |
| `OCISodaReplOneAndGetWithCtnt()` | Replaces a document in a collection and returns the result document. The input document is represented only by the content string. |
| `OCISodaReplOneAndGetWithKey()` | Replace a document in a collection given a key and returns the result document. |
| `OCISodaReplOneWithCtnt()` | Replaces a document in a collection. The document is represented only by the content string |
| `OCISodaReplOneWithKey()` | Replaces a document in a collection given a key. |
| `OCISodaOperKeysSet()` | Sets the array of keys on Operation Options Handle. |

## 36.2.1 OCISodaBulkInsert()

Inserts an array of documents into a collection.

### Purpose

To insert an array of documents into a collection.

### Syntax

```
sword OCISodaBulkInsert(OCISvcCtx           *svchp,
                        OCISodaColl         *collection,
                        OCISodaDoc         **documentarray,
                        ub4                  arraylen,
                        OCISodaOutputOptions *opoptns,
                        OCIError            *errhp,
                        ub4                  mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `collection (IN)` | The collection handle for an existing collection. |
| `documentarray (IN)` | Array of handles of documents to be inserted into the collection. |

| Parameter | Description |
|---|---|
| `arrarylen (IN)` | The length of the document handle array supplied in the `documentarray` parameter. |
| `opoptns (OUT)` | Output options handle. Valid attributes are:<br>• `OCI_ATTR_SODA_DOC_COUNT.` — The number of documents inserted.<br>**Note**: If an error occurs, `OCI_ATTR_SODA_DOC_COUNT` contains the number of documents inserted before the error occurred. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if the document array insertion into the specified collection is successful.

`OCI_ERROR`, if the document array insertion failed. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Inserting Documents into Collections with SODA for C

## 36.2.2 OCISodaBulkInsertAndGet()

Inserts an array of documents into a collection, and returns an array of result documents each containing all document components except for the content.

### Purpose

To insert an array of documents into a collection, and return an array of result documents, each containing all document components except for the content. The components generated by SODA during the insert, such as key (if the collection has auto-assigned keys), last-modified timestamp, created-on timestamp, and version, are returned with each result document.

### Syntax

```
sword OCISodaBulkInsertAndGet(OCISvcCtx           *svchp,
                              OCISodaColl         *collection,
                              OCISodaDoc         **documentarray,
                              ub4                   arraylen,
                              OCISodaOutputOptions *opoptns,
                              OCIError            *errhp,
                              ub4                   mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `collection (IN)` | The collection handle for an existing collection. |
| `documentarray (IN/OUT)` | Array of handles of documents to be inserted into the collection.<br>**Note**: Since the result documents are returned using the same parameter, you need to save a reference to each input document handle so that you can free it later along with each result document handle, using `OCIHandleFree()` method. If you do not save a reference to each input document handle, it is overwritten with a reference to the result document, and the input document handle will never be properly freed (leads to memory leak). |
| `arrarylen (IN)` | The length of the document handle array supplied in `documentarray` parameter. |
| `opoptns (OUT)` | Output options handle. Valid attributes are:<br>- `OCI_ATTR_SODA_DOC_COUNT`: The number of documents inserted. If an error occurs, `OCI_ATTR_SODA_DOC_COUNT` contains the number of documents inserted before the occurrence of the error. |
| `errhp (OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

**Returns**

`OCI_SUCCESS`, if the document array insertion into the specified collection is successful.

`OCI_ERROR`, if the document array insertion failed. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Inserting Documents into Collections with SODA for C.

## 36.2.3 OCISodaBulkInsertAndGetWithCtnt()

Inserts an array of documents into a collection using only their contents, and returns an array of result documents containing all the document components except for the content.

**Purpose**

To insert an array of documents into a collection using only their contents and returns an array of result documents, each document containing all document components except for the content.

> **Note:**
>
> This is a convenience method for inserting JSON documents only without constructing document handles for them. The more general method is `OCISodaBulkInsertAndGet`.

### Syntax

```
sword OCISodaBulkInsertAndGetWithCont(OCISvcCtx            *svchp,
                                      OCISodaColl          *collection,
                                      void
**contentStrings,
                                      ub4
*contentStringLengths,
                                      OraText             **keys,
                                      ub4                  *keyLengths,
                                      ub4                   arrayLength,
                                      ub4                   docFlags,
                                      OCISodaDoc          **documents,
                                      OCISodaOutputOptions  *opoptns,
                                      OCIError             *errhp
                                      ub4                   mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `collection (IN)` | The collection handle for an existing collection. |
| `contentStrings (IN)` | An array of strings that contains the JSON content for the documents.<br>**Note**: This method works only for JSON documents. |
| `contentStringLengths (IN)` | An array of content string lengths. |
| `keys (IN)` | An array of keys for the inserted documents. Provide the keys, if the collection has client-assigned keys. Otherwise, set to null. |
| `keyLengths (IN)` | An array of key lengths. Needs to be set only if the collection has client-assigned keys. Otherwise, must be set to null. |
| `arrayLength (IN)` | The length of the `contentStrings` and `keys` arrays. |

| Parameter | Description |
|---|---|
| `docFlags (IN)` | The encoding flags used to indicate the encoding of the supplied document's content. Valid flag options are:<br>• `OCI_DEFAULT` — If you use this parameter value then you are, in effect, declaring that the document content is in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle). If that is not the case for a given document, then the result of trying to write the document is unpredictable.<br>• `OCI_SODA_DETECT_JSON_ENC` — Automatically detects the encoding of the document content as either UTF-8, UTF-16 LE, or UTF-16 BE. If you use this parameter value then you are, in effect, declaring that the document content is either UTF-8, UTF-16 LE, or UTF-16 BE. If that is not the case for a given document, then the result of trying to write the document is unpredictable. |
| `documents (OUT)` | Returns an array of result documents containing all document components except for the content. Each result document contains the components such as key (if the collection has auto-assigned keys), last-modified timestamp, created-on timestamp, and version generated by SODA insert command. |
| `opoptns (OUT)` | Specifies the output options handle. Valid attributes are:<br>• `OCI_ATTR_SODA_DOC_COUNT.` — The number of documents inserted.<br>**Note**: If an error occurs, `OCI_ATTR_SODA_DOC_COUNT` contains the number of documents inserted before the error occurred. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

**Returns**

`OCI_SUCCESS`, if the document insertion into the specified collection is successful.

`OCI_ERROR`, if the document insertion failed. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Inserting Documents into Collections with SODA for C.

## 36.2.4 OCISodaBulkInsertWithCtnt()

Inserts an array of documents into a collection using only their contents.

**Purpose**

To insert an array of documents into a collection using only their contents.

> **Note:**
>
> This is a convenience method for inserting JSON documents only without constructing document handles for them. The more general method is `OCISodaBulkInsert`.

**Syntax**

```
sword OCISodaBulkInsertWithCtnt(OCISvcCtx            *svchp,
                                OCISodaColl          *collection,
                                void                **contentStrings,
                                ub4                  *contentStringLengths,
                                OraText             **keys,
                                ub4                  *keyLengths,
                                ub4                   arrayLength,
                                ub4                   docFlags,
                                OCISodaOutputOptions *opoptns,
                                OCIError             *errhp,
                                ub4                   mode);
```

**Parameters**

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `collection (IN)` | The collection handle for an existing collection. |
| `contentStrings (IN)` | An array of strings that contain the JSON content for the documents. This method works only for JSON documents. |

| Parameter | Description |
|---|---|
| `contentStringLengths (IN)` | An array of content string lengths. |
| `keys (IN)` | An array of keys for the inserted documents. Provide the keys, if the collection has client-assigned keys. Otherwise, set to null. |
| `keyLengths (IN)` | An array of key lengths. Needs to be set only if the collection has client-assigned keys. Otherwise, must be set to null. |
| `arrayLength (IN)` | The length of the `contentStrings` and `keys` arrays. |
| `docFlags (IN)` | The encoding flags used to create the documents. Valid flag options are:<br>• `OCI_DEFAULT` — If you use this parameter value then you are, in effect, declaring that the document content is in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle). If that is not the case for a given document, then the result of trying to write the document is unpredictable.<br>• `OCI_SODA_DETECT_JSON_ENC` — Automatically detects the encoding of the document content as either UTF-8, UTF-16 LE, or UTF-16 BE. If you use this parameter value then you are, in effect, declaring that the document content is either UTF-8, UTF-16 LE, or UTF-16 BE. If that is not the case for a given document, then the result of trying to write the document is unpredictable. |
| `opoptns (OUT)` | Specifies the output options handle. Valid attributes are:<br>• `OCI_ATTR_SODA_DOC_COUNT.` — The number of documents inserted.<br>**Note**: If an error occurs, `OCI_ATTR_SODA_DOC_COUNT` contains the number of documents inserted before the error occurred. |
| `errhp (OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

**Returns**

`OCI_SUCCESS`, if the document insertion into the specified collection is successful.

`OCI_ERROR`, if the document insertion failed. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For example, see Inserting Documents into Collections with SODA for C

## 36.2.5 OCISodaCollCreate()

Creates a new document collection with default metadata settings.

**Purpose**

Create a new document collection with default metadata settings.

**Syntax**

```
sword OCISodaCollCreate(OCISvcCtx       *svchp,
                        const OraText   *collname,
                        ub4              collnamelen,
                        OCISodaColl    **collection,
                        OCIError        *errhp,
                        ub4              mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `collname (IN)` | The name of the collection. |
| `collnamelen (IN)` | The length of the collection name. |
| `collection (OUT)` | Allocates the collection handle representing the document collection that was created. Call `OCIHandleFree()` to free the collection handle when it is no longer required. |
| `errhp (IN/OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` - When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>• `OCI_SODA_COLL_CREATE_MAP` - Creates a new collection mapping to an existing table. Minimal checking is performed to ensure that the table has correct shape matching the default collection metadata. This function returns an error, if this check fails. |

### Returns

`OCI_SUCCESS`, if the collection is successfully created.

`OCI_ERROR`, if the collection is not created. The `OCIError` parameter has the necessary error information.

### Usage Notes

- If the collection with the specified name already exists, then this function acts as `OCISodaCollOpen()` and opens the existing collection.
- This function, unlike a SQL DDL operation, does not perform a commit operation before or after it runs. (Unless you specify `OCI_SODA_ATOMIC_COMMIT` mode, only then is the current transaction committed after the operation runs).

### Examples

**Example 36-1 Creating a Collection**

for examples, see Creating a Document Collection for SODA with C.

## 36.2.6 OCISodaCollCreateWithMetadata()

Creates a new document collection with metadata.

### Purpose

Create a new document collection with metadata.

### Syntax

```
sword OCISodaCollCreateWithMetadata(OCISvcCtx      *svchp,
                                    const OraText  *collname,
                                    ub4             collnamelen,
                                    OraText        *metadata,
                                    ub4             metadatalen,
                                    OCISodaColl   **collection,
                                    OCIError       *errhp,
                                    ub4             mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `collname (IN)` | The name of the collection. |
| `collnamelen (IN)` | The length of the collection name. |
| `metadata (IN)` | The JSON string that contains the collection metadata information. If `metadata` is `NULL`, it uses the default metadata settings to create the collection. |
| `metadatalen (IN)` | The length of the metadata JSON string. |
| `collection (OUT)` | Allocates the collection handle representing the document collection that was created. Call `OCIHandleFree()` to free the collection handle when it is no longer required. |
| `errhp (IN/OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` - Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` - When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>• `OCI_SODA_COLL_CREATE_MAP` - Creates a new collection mapping to an existing table. Minimal checking is performed to ensure that the table has correct shape, matching the specified collection metadata or default metadata, if metadata parameter is `NULL`. This function returns an error, if this check fails. |

### Returns

`OCI_SUCCESS`, if the collection is successfully created.

`OCI_ERROR`, if the collection is not created. The `OCIError` parameter has the necessary error information.

### Usage Notes

- If the collection with the specified name already exists and that collection has metadata equivalent to the supplied metadata, then this function acts as `OCISodaCollOpen()` and opens the existing collection. If the collection contains metadata that does not match with the specified collection name, then an error is returned.
- This function, unlike a SQL DDL operation, does not perform a commit operation before or after it runs. (Unless you specify `OCI_SODA_ATOMIC_COMMIT` mode, only then is the current transaction committed after the operation runs).

### Examples

For example, see Creating a Document Collection for SODA with C.

## 36.2.7 OCISodaCollDrop()

Drops a collection from the database.

### Purpose

Drop a collection from the database.

### Syntax

```
sword OCISodaCollDrop(OCISvcCtx   *svchp,
                      OCISodaColl *coll,
                      boolean     *isDropped,
                      OCIError    *errhp,
                      ub4          mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `coll (IN)` | The collection to be dropped. If `coll` no longer references an existing collection, then no error is returned, but `isDropped` is `FALSE` indicating the drop operation was not successful after the invocation of `OCISodaCollDrop()`. |
| `isDropped (OUT)` | Returns the status of the drop operation: `TRUE` if the drop operation is successful, `FALSE` if the drop operation is not successful. |
| `errhp (IN/OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if dropping the collection is successful.

`OCI_ERROR`, if dropping the collection failed. The `OCIError` parameter has the necessary error information.

### Usage Notes

- This function only drops the collection; it does not free the collection handle. Call `OCIHandleFree()` to free the collection handle to avoid memory leaks.
- This function, unlike a SQL DDL operation, does not perform a commit operation before or after it runs. (Unless you specify `OCI_SODA_ATOMIC_COMMIT` mode, only then is the current transaction committed after the operation runs).

### Examples

For examples, see Dropping a Document Collection with SODA for C.

## 36.2.8 OCISodaCollGetNext()

Gets the next collection from the collection cursor.

### Purpose

Get the next collection from the collection cursor.

### Syntax

```
sword OCISodaCollGetNext(OCISvcCtx                *svchp,
                         const OCISodaCollCursor  *cur,
                         OCISodaColl             **coll,
                         OCIError                 *errhp,
                         ub4                       mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `cur (IN)` | The collection cursor handle remains valid when allocated by `OCISodaCollList()`. Call `OCIHandleFree()` to free the collection cursor handle when it is no longer required. |
| `coll (OUT)` | Allocates the collection handle and returns it to the next collection to be found. Call `OCIHandleFree()` to free the collection handle when it is no longer required. |
| `errhp (IN/OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes. |

### Returns

`OCI_SUCCESS`, if the next collection is returned from the specified cursor.

`OCI_ERROR`, if there was an error fetching the next collection. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Discovering Existing Collection with SODA for C

## 36.2.9 OCISodaCollList()

Lists the collections in the schema associated with the supplied service context handle parameter.

**Purpose**

List the collections in the schema.

**Syntax**

```
sword OCISodaCollList(OCISvcCtx          *svchp,
                      const OraText      *startname,
                      ub4                 stnamelen,
                      OCISodaCollCursor **cur,
                      OCIError           *errhp,
                      ub4                 mode);
```

**Parameters**

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `startname (IN)` | The collection start name. Can be set to `NULL` and `stnamelen` set to the value 0 to return all existing collections. |
| `stnamelen (IN)` | The length of the collection start name. |
| `cur (OUT)` | Allocates the collection cursor handle and returns it. The collection cursor represents the collection list. Call `OCIHandleFree()` to free the collection cursor handle when it is no longer required. |
| `errhp (IN/OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>- `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>- `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>**Note:**<br>`OCISodaCollList()` does not make any changes to the database; however, you can use this mode to commit an existing transaction without performing another round trip. |

**Returns**

`OCI_SUCCESS`, if getting a cursor over all collections in the schema is successful.

`OCI_ERROR`, if getting a cursor over all collections in the schema is not successful. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Discovering Existing Collection with SODA for C.

## 36.2.10 OCISodaCollOpen()

Opens the collection.

**Purpose**

Open the collection.

### Syntax

```
sword OCISodaCollOpen(OCISvcCtx      *svchp,
                      const OraText  *collname,
                      ub4             collnamelen,
                      OCISodaColl   **coll,
                      OCIError       *errhp,
                      ub4             mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `collname (IN)` | The name assigned to the collection. |
| `collnamelen (IN)` | The length of the collection name. |
| `coll (OUT)` | Allocates the collection handle and returns it with the specified collection name. The function returns `OCI_SUCCESS` if there are no errors while trying to open the collection. If the collection with the supplied name is not present, that is not considered an error and returns `OCI_SUCCESS`, but the collection-handle pointer returns as `NULL`. So a way to check if the collection exists and was successfully opened, is to check that the `coll (OUT)` pointer is not `NULL` after this function returns with `OCI_SUCCESS`. Call `OCIHandleFree()` to free the collection handle when it is no longer required. |
| `errhp (IN/OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>**Note:**<br>`OCISodaCollOpen()` does not make any changes to the database; however, you can use this mode to commit an existing transaction without performing another round trip. |

**Returns**

`OCI_SUCCESS`, if there were no errors while trying to open the collection.

`OCI_ERROR`, if errors were encountered while trying to open the collection. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Opening an Existing Document Collection with SODA for C.

## 36.2.11 OCISodaDataGuideGet()

Gets JSON data guide information.

**Purpose**

To get the JSON data guide information.

### Syntax

```
sword OCISodaDataGuideGet(OCISvcCtx          *svchp,
                          const OCISodaColl  *collection,
                          ub4                 docFlags,
                          OCISodaDoc        **doc,
                          OCIError           *errhp,
                          ub4                 mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `collection(IN)` | The collection handle. |
| `docFlags (IN)` | The following flags control the encoding of the returned document's content. Valid flag options are:<br>• `OCI_DEFAULT` — Document content is returned in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle).<br>• `OCI_SODA_AS_AL32UTF8` — Document content is returned in the character set AL32UTF8. |
| `doc (OUT)` | The document handle with content representing the JSON data guide. |
| `errhp (OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>**Note:**<br>`OCISodaDataGuideGet()` does not make any changes to the database; however, you can use this mode to commit an existing transaction without performing another round trip. |

**Returns**

`OCI_SUCCESS`, if getting the data guide information of a specified collection is successful.

`OCI_ERROR`, if getting data guide information results in an error. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Getting a Data Guide for a Collection with SODA for C.

## 36.2.12 OCISodaDocCount()

Counts the number of documents matching the attributes set in the operation options handle. If the operation options handle has no attributes set, then the function counts all the documents in the collection.

**Purpose**

To count the number of documents matching the attributes set in the operation options handle. If the operation options handle has no attributes set, then the function counts all the documents in the collection.

### Syntax

```
sword OCISodaDocCount(OCISvcCtx                     *svchp,
                      const OCISodaColl             *coll,
                      const OCISodaOperationOptions *optns,
                      ub8                           *numdocs,
                      OCIError                      *errhp,
                      ub4                            mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The collection handle for an existing collection. |
| `optns (IN)` | The operation options handle. If skip and limit attributes are set on the operation options handle, then an error is returned. |
| `numdocs (OUT)` | The number of documents in the collection matching attributes (if any) set on the operation options handle. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>**Note:**<br>`OCISodaDocCount()` does not make any changes to the database; however, you can use this mode to commit an existing transaction without performing another round trip. |

### Returns

`OCI_SUCCESS`, if the document count from the specified collection is successful.

`OCI_ERROR`, if the document count failed due to hard errors. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Counting the Number of Documents Found.

## 36.2.13 OCISodaDocCountWithFilter()

Counts the number of documents matching a filter specification.

### Purpose

To count the number of documents matching a filter specification.

> **Note:**
>
> This is a convenience method for counting documents matching a filter specification. A more general method for counting documents is `OCISodaDocCount()` which can count documents matching a filter specification or other criteria.

### Syntax

```
sword OCISodaDocCountWithFilter(OCISvcCtx        *svchp,
                                const OCISodaColl *coll,
                                const OraText     *filterSpec,
                                ub4                filterSpecLen,
                                ub8               *numdocs,
                                OCIError          *errhp,
                                ub4                mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The collection handle for an existing collection. |
| `filterSpec (IN)` | The JSON filter string used to search the documents in a collection. |
| `filterSpecLen (IN)` | The length of the filter string. |
| `numdocs (OUT)` | The number of documents in the collection matching the filter. |
| `errhp (OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>- `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>- `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>**Note:**<br>`OCISodaDocCountWithFilter()` does not make any changes to the database; however, you can use this mode to commit an existing transaction without performing another round trip. |

### Returns

`OCI_SUCCESS`, if the document count from the specified collection is successful.

`OCI_ERROR`, if the document count failed due to hard errors. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

## 36.2.14 OCISodaDocCreate()

Creates a document.

### Purpose

Create a document.

### Syntax

```
sword OCISodaDocCreate (OCIEnv          *envhp,
                        const void      *content,
                        ub4              contentLength,
                        ub4              docFlags,
                    OCISodaDoc       **document,
                    OCIError          *errhp,
                    ub4                mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `envhp (IN)` | The environment handle. |
| `content(IN)` | A string containing the document content. |
| `contentLength (IN)` | The string length of the content parameter. |
| `docFlags (IN)` | Used to detect JSON encoding while creating a document. Valid flag options are:<br>• `OCI_DEFAULT` — If you use this parameter value then you are, in effect, declaring that the document content is in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle). If that is not the case for a given document, then the result of trying to write the document is unpredictable.<br>• `OCI_SODA_DETECT_JSON_ENC` — Automatically detects the encoding of the document content as either UTF-8, UTF-16 LE, or UTF-16 BE. If you use this parameter value then you are, in effect, declaring that the document content is either UTF-8, UTF-16 LE, or UTF-16 BE. If that is not the case for a given document, then the result of trying to write the document is unpredictable. |
| `document (OUT)` | Allocates and returns the document handle. Call `OCIHandleFree()` to free the document handle when it is no longer required. |
| `errhp (IN/OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes. |

### Returns

`OCI_SUCCESS`, if the document is successfully created.

`OCI_ERROR`, if the document creation failed. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

**Example 36-2 Creating a Document**

For examples, see Creating Documents with SODA for C.

## 36.2.15 OCISodaDocCreateWithKey()

Creates a document with a document key and content.

### Purpose

Create a document with a document key and content.

### Syntax

```
sword OCISodaDocCreateWithKey(OCIEnv         *envhp,
                              const void     *content,
                              ub4             contentLength,
                              const OraText  *key,
                              ub4             keylen,
                              ub4             docFlags,
                              OCISodaDoc    **document,
                              OCIError       *errhp,
                              ub4             mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `envhp (IN)` | The environment handle. |
| `content (IN)` | A string containing the document content. |
| `contentLength (IN)` | The string length of the content parameter. |
| `key (IN)` | A string that contains the document key. The key can be `NULL`.<br>For example, in the default case, collection has auto-generated keys, so you do not want to set the key on the document (if you do, you get an error when you pass such a doc to a write operation, such as insert). You only want to set a non-`NULL` key if your collection is configured with client-assigned keys, which is not the default. |
| `keylen (IN)` | The length of the key string. |

| Parameter | Description |
|---|---|
| `docFlags (IN)` | The flags used to create a document. Valid flag options are:<br>• `OCI_DEFAULT` — If you use this parameter value then you are, in effect, declaring that the document content is in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle). If that is not the case for a given document, then the result of trying to write the document is unpredictable.<br>• `OCI_SODA_DETECT_JSON_ENC` — Automatically detects the encoding of the document content as either UTF-8, UTF-16 LE, or UTF-16 BE. If you use this parameter value then you are, in effect, declaring that the document content is either UTF-8, UTF-16 LE, or UTF-16 BE. If that is not the case for a given document, then the result of trying to write the document is unpredictable. |
| `document (OUT)` | Allocates and returns the document handle. Call `OCIHandleFree()` to free the document handle when it is no longer required. |
| `errhp (IN/OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes. |

**Returns**

`OCI_SUCCESS`, if the document is successfully created.

`OCI_ERROR`, if the document creation failed. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Creating Documents with SODA for C.

## 36.2.16 OCISodaDocCreateWithKeyAndMType()

Creates a document with a document key, content, and media type.

**Purpose**

Create a document with a document key, content, and media type.

### Syntax

```
sword OCISodaDocCreateWithKeyAndMType(OCIEnv              *envhp,
                                      const void          *content,
                                      ub4                  contentLength,
                                      const OraText       *key,
                                      ub4                  keylen,
                                      const OraText       *mediaType,
                                      ub4
mediaTypeLength,
                                      ub4                  docFlags,
                                      OCISodaDoc         **document,
                                      OCIError            *errhp,
                                      ub4                  mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `envhp (IN)` | The environment handle. |
| `content (IN)` | A string containing the document content. |
| `contentLength (IN)` | The string length of the content parameter. |
| `key (IN)` | A string that contains the document key. The key can be `NULL`.<br>For example, in the default case, collection has auto-generated keys, so you do not want to set the key on the document (if you do, you get an error when you pass such a doc to a write operation, such as insert). You only want to set a non-`NULL` key if your collection is configured with client-assigned keys, which is not the default. |
| `keylen (IN)` | The length of the key string. |
| `mediaType (IN)` | A string that contains the document media type. If no media type value is specified, it defaults to "`application/json`". By specifying a value, you can create non-JSON documents (using a media type other than "`application/json`"). |
| `mediaTypeLength (IN)` | The length of the media type string. |

| Parameter | Description |
|---|---|
| `docFlags (IN)` | The flags used to create the document. Valid flag options are:<br>• `OCI_DEFAULT` — If you use this parameter value then you are, in effect, declaring that the document content is in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle). If that is not the case for a given document, then the result of trying to write the document is unpredictable.<br>• `OCI_SODA_DETECT_JSON_ENC` — Automatically detects the encoding of the document content as either UTF-8, UTF-16 LE, or UTF-16 BE. If you use this parameter value then you are, in effect, declaring that the document content is either UTF-8, UTF-16 LE, or UTF-16 BE. If that is not the case for a given document, then the result of trying to write the document is unpredictable. |
| `document (OUT)` | Allocates and returns the document handle. Call `OCIHandleFree()` to free the document handle when it is no longer required. |
| `errhp (IN/OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes. |

### Returns

`OCI_SUCCESS`, if the document is successfully created.

`OCI_ERROR`, if the document creation failed. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Creating Documents with SODA for C.

## 36.2.17 OCISodaDocGetNext()

Gets the next document from the cursor.

### Purpose

To get the next document from the cursor.

### Syntax

```
sword OCISodaDocGetNext(OCISvcCtx                *svchp,
                        const OCISodaDocCursor   *cursor,
                        OCISodaDoc              **doc,
                        OCIError                 *errhp,
                        ub4                       mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `cursor (IN)` | The cursor for the document results. |
| `doc (OUT)` | The document handle returned by the cursor. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — the default mode meaning calling this function executes its statement. |

### Returns

`OCI_SUCCESS`, if the document is found in the specified collection.

`OCI_ERROR`, if the document search failed. The `OCIError` parameter has the necessary error information.

`OCI_NO_DATA`, if data not found. (exit criteria).

### Usage Notes

None.

### Examples

for examples, see Finding Documents in Collections with SODA for C.

## 36.2.18 OCISodaFind()

Finds the documents in a collection matching the attributes set in operation options input handle. If no conditions are set, all the documents in the collection are returned.

### Purpose

To find the documents in a collection matching the attributes set in operation options input handle. If no conditions are set, then all the documents in the collection are returned.

### Syntax

```
sword OCISodaFind(OCISvcCtx                      *svchp,
                  const OCISodaColl              *coll,
                    const OCISodaOperationOptions  *findOptions,
                    ub4                             docFlags,
                    OCISodaDocCursor              **cursor,
                    OCIError                       *errhp,
                    ub4                             mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The collection to search for the documents. |
| `findOptions (IN)` | The operation options handle with attributes to drive the find operation. If no attributes are set, then all the documents in the collection are returned. |
| `docFlags (IN)` | The following flags control the encoding of the returned document's content. Valid flag options are<br>• `OCI_DEFAULT` — Document content is returned in the character set defined by the environment handle (or environment variable NLS_LANG, if not set for the handle).<br>• `OCI_SODA_AS_STORED` — Document content is returned in the character set in which the content was stored in the database. Use this parameter value only for BLOB content. If the content is not stored in a BLOB instance then an error is raised.<br>• `OCI_SODA_AS_AL32UTF8` — Document content is returned in the character set `AL32UTF8`. |
| `cursor (OUT)` | The cursor for the document results. |
| `errhp (OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>**Note:**<br>`OCISodaFindOneWithKey()` does not make any changes to the database; however, you can use this mode to commit an existing transaction without performing another round trip. |

### Returns

`OCI_SUCCESS`, if the document is found in the specified collection.

`OCI_ERROR`, in case of hard errors. The `OCIError` parameter has the necessary error information.

`OCI_NO_DATA`, if data not found. (exit criteria).

### Usage Notes

None.

### Examples

For examples, see Finding Documents in Collections with SODA for C.

## 36.2.19 OCISodaFindOne()

Finds the document in a collection matching the attributes set in operation options input handle. If more than one document in the collection matches the attributes set, then the first document is returned.

### Purpose

To find the document in a collection matching the attributes set in operation options input handle. If more than one document in the collection matches the attributes set, then the first document is returned.

### Syntax

```
sword OCISodaFindOne(OCISvcCtx                          *svchp,
                     const OCISodaColl                  *coll,
                     const OCISodaOperationOptions  *findOptions,
                     ub4                                 docFlags,
                     OCISodaDoc                        **doc,
                     OCIError                           *errhp,
                     ub4                                 mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The collection to search for the documents. |
| `findOptions (IN)` | The operation options handle with attributes to drive the find operation. If more than one document in the collection matches, then the first matching document is returned. |
| `docFlags (IN)` | The following flags control the encoding of the returned document's content. Valid flag options are:<br>• `OCI_DEFAULT` — Document content is returned in the character set defined by the environment handle (or environment variable NLS_LANG, if not set for the handle).<br>• `OCI_SODA_AS_STORED` — Document content is returned in the character set in which the content was stored in the database. Use this parameter value only for BLOB content. If the content is not stored in a `BLOB` instance then an error is raised.<br>• `OCI_SODA_AS_AL32UTF8` —Document content is returned in the character set `AL32UTF8`. |
| `doc (OUT)` | The document found (returns `NULL` if no document is found). |
| `errhp (OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>**Note:**<br>`OCISodaFindOneWithKey()` does not make any changes to the database; however, you can use this mode to commit an existing transaction without performing another round trip. |

**Returns**

`OCI_SUCCESS`, if the document is found in the specified collection.

`OCI_ERROR`, if the document search failed. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Finding Documents in Collections with SODA for C.

## 36.2.20 OCISodaFindOneWithKey()

Finds a single document in a collection, given a key.

**Purpose**

To find a single document in a collection, given a key.

> **Note:**
>
> This is a convenience method for finding a document by a key. A more general method is `OCISodaFindOne`.

### Syntax

```
sword OCISodaFindOneWithKey(OCISvcCtx          *svchp,
                            const OCISodaColl  *coll,
                            const OraText      *key,
                            ub4                 keylen,
                            ub4                 docFlags,
                            OCISodaDoc        **doc,
                            OCIError           *errhp,
                            ub4                 mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `coll (IN)` | The collection to search for the document. |
| `key (IN)` | The key used to identify the document. |
| `keylen (IN)` | The length of the key. |
| `docFlags (IN)` | The following flags control the encoding of the returned document's content. Valid flag options are:<br>• `OCI_DEFAULT` — Document content is returned in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle).<br>• `OCI_SODA_AS_STORED` — Document content is returned in the character set in which the content was stored in the database. Use this parameter value *only* for `BLOB` content. If the content is not stored in a BLOB instance then an error is raised.<br>• `OCI_SODA_AS_AL32UTF8` — Document content is returned in the character set AL32UTF8. |
| `doc (OUT)` | Allocates and returns the document handle if the document is found in the collection. If the document is not found in the collection, a `NULL` document is returned and the function returns `OCI_NO_DATA`. Call `OCIHandleFree()` to free the collection handle when the find operation completes. |
| `errhp (IN/OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back.<br>**Note:**<br>`OCISodaFindOneWithKey()` does not make any changes to the database; however, you can use this mode to commit an existing transaction without performing another round trip. |

### Returns

`OCI_SUCCESS`, if the document is found in the specified collection.

`OCI_NO_DATA`, if the document was not found.

`OCI_ERROR`, if there was an error in trying to find the document. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Finding Documents in Collections with SODA for C

## 36.2.21 OCISodaIndexCreate()

Creates an index with an index specification.

### Purpose

To create an index with an index specification. The following three types of index specifications can be supplied, B-tree, JSON search index, and spatial.

### Syntax

```
sword OCISodaIndexCreate(OCISvcCtx         *svchp,
                         const OCISodaColl *coll,
                         const OraText     *indexspec,
                         ub4                speclen,
                         OCIError          *errhp,
                         ub4                mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The handle for the collection to be indexed. |
| `indexspec (IN)` | The index specification. |
| `speclen(IN)` | The index specification length. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` - Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` - When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if index creation on the specified collection is successful.

`OCI_ERROR`, if index creation results in an error. The `OCIError` parameter has the necessary error information.

### Usage Notes

This function, unlike a SQL DDL operation, does not perform a commit operation before or after it runs. (Unless you specify OCI_SODA_ATOMIC_COMMIT mode, only then is the current transaction committed after the operation runs).

### Examples

For examples, see Indexing the Documents in a Collection with SODA for C.

### Related Topics

- Overview of SODA Indexing

- SODA Index Specifications (Reference)

## 36.2.22 OCISodaIndexDrop()

Drops an index.

**Purpose**

To drop an index.

**Syntax**

```
sword OCISodaIndexDrop(OCISvcCtx   *svchp,
                       OraText     *indexname,
                       ub4          indexnamelen,
                       boolean     *isDropped,
                       OCIError    *errhp,
                       ub4          mode);
```

**Parameters**

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `indexname (IN)` | The index name. |
| `indexnamelen(IN)` | The index name length. |
| `isDropped (OUT)` | Returns the status of the drop operation: `TRUE` if the drop operation is successful, `FALSE` if the index with the specified name does not exist. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_INDEX_DROP_FORCE` - Forces the index to be dropped. Should only be used for JSON Search Index or Spatial Index. Not supported for dropping B-tree indexes.<br>**Note**: See DROP INDEX for more information about FORCE flag.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

**Returns**

`OCI_SUCCESS`, if dropping an index on specified collection is successful.

`OCI_ERROR`, if dropping an index results in an error. The `OCIError` parameter has the necessary error information.

**Usage Notes**

This function, unlike a SQL DDL operation, does not perform a commit operation before or after it runs. (Unless you specify `OCI_SODA_ATOMIC_COMMIT` mode, only then is the current transaction committed after the operation runs).

**Examples**

For example, see Dropping an Index with SODA for C.

## 36.2.23 OCISodaInsert()

Inserts a document into a collection.

**Purpose**

Insert a document into a collection.

**Syntax**

```
sword OCISodaInsert(OCISvcCtx   *svchp,
                    OCISodaColl *collection,
                    OCISodaDoc  *document,
                    OCIError    *errhp,
                    ub4          mode);
```

**Parameters**

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `collection (IN)` | The collection handle for an existing collection. |
| `document (IN)` | The handle of the document that is to be inserted into the collection. |
| `errhp (IN/OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

**Returns**

`OCI_SUCCESS`, if the document insertion into the specified collection is successful.

`OCI_ERROR`, if the document insertion failed. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Inserting Documents into Collections with SODA for C.

## 36.2.24 OCISodaInsertAndGet()

Inserts a document into a collection, returning a result document containing all document components except for content.

**Purpose**

Insert a document into a collection, returning a result document containing all document components except for content. The components generated by SODA during the insert, such as key (if the collection has auto-assigned keys), last-modified timestamp, created-on timestamp, and version, are returned as part of the result document.

**Syntax**

```
sword OCISodaInsertAndGet(OCISvcCtx      *svchp,
                          OCISodaColl    *collection,
                          OCISodaDoc    **document,
                          OCIError       *errhp,
                          ub4             mode);
```

**Parameters**

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `collection (IN)` | The collection handle for an existing collection. |
| `document (IN/OUT)` | The handle of the document that is to be inserted into the collection. Returns the result document through this same parameter as it is an `IN/OUT`.<br>**Caution:**<br>Because the result document is returned using the same parameter, you need to save a reference to the input document handle so that you can free it later, along with the result document handle, using `OCIHandleFree()`. If you do not save a reference to the input document handle, it is overwritten with a reference to the result document, and the input document handle will never be properly freed (it's a memory leak). |
| `errhp (IN/OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

**Returns**

`OCI_SUCCESS`, if the document insertion into the specified collection is successful.

`OCI_ERROR`, if the document insertion failed. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Inserting Documents into Collections with SODA for C.

## 36.2.25 OCISodaInsertAndGetWithCtnt()

Inserts a document into a collection using just the content, returning a result document containing all document components except for content.

**Purpose**

Insert a document into a collection using just the content, returning a result document containing all document components except for content. The components generated by SODA during the insert, such as key (if the collection has auto-assigned keys), last-modified timestamp, created-on timestamp, and version, are returned as part of the result document.

> **Note:**
>
> This is a convenience method for inserting a JSON document without constructing a document handle for them. A more general method is `OCISodaInsertAndGet`.

### Syntax

```
sword OCISodaInsertAndGetWithCtnt(OCISvcCtx        *svchp,
                                  OCISodaColl      *collection,
                                  const OraText    *key,
                                  ub4               keyLength,
                                  const void       *content,
                                  ub4               contentLength,
                                  ub4               docFlags,
                                  OCISodaDoc      **document,
                                  OCIError         *errhp,
                                  ub4               mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `collection (IN)` | The collection handle for an existing collection. |
| `key (IN)` | A string that contains the document key. Must be `NULL` if inserting a document into a collection configured for auto-generated keys. |
| `keyLength (IN)` | The length of the key string. Must be 0 if inserting a document into a collection configured for auto-generated keys. |
| `content (IN)` | A string containing the JSON content for the document.<br>**Note**: This method works only for JSON documents. |
| `contentLength (IN)` | The string length of the `content` parameter. |
| `docFlags (IN)` | The encoding flags used to indicate the encoding of the supplied document's content. Valid flag options are:<br>• `OCI_DEFAULT` — If you use this parameter value then you are, in effect, declaring that the document content is in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle). If that is not the case for a given document, then the result of trying to write the document is unpredictable.<br>• `OCI_SODA_DETECT_JSON_ENC` — Automatically detects the encoding of the document content as either UTF-8, UTF-16 LE, or UTF-16 BE. If you use this parameter value then you are, in effect, declaring that the document content is either UTF-8, UTF-16 LE, or UTF-16 BE. If that is not the case for a given document, then the result of trying to write the document is unpredictable. |

| Parameter | Description |
|---|---|
| `document (OUT)` | Allocates and returns a result document containing all document components except for content. The components generated by SODA during the insert, such as key (if the collection has auto-assigned keys), last-modified timestamp, created-on timestamp, and version, are returned as part of the result document.<br>If `NULL` is passed for this parameter, then the result document is not returned. (In other words, the function then behaves identically to the `OCISodaInsertWithCtnt()` function, which does not return a result document). |
| `errhp (IN/OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if the document insertion into the specified collection is successful.

`OCI_ERROR`, if the document insertion failed. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Inserting Documents into Collections with SODA for C

## 36.2.26 OCISodaInsertWithCtnt()

Inserts a document into a collection using only the content.

### Purpose

Insert a document into a collection using only the content.

**Note:**

This is a convenience method for inserting a JSON document without constructing a document handle for them. A more general method is `OCISodaInsert`.

### Syntax

```
sword OCISodaInsertWithCtnt(OCISvcCtx          *svchp,
                            OCISodaColl        *collection,
                            const OraText      *key,
                            ub4                 keyLength,
                            const void         *content,
                            ub4                 contentLength,
                            ub4                 docFlags,
                            OCIError           *errhp,
                            ub4                 mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `collection (IN)` | The collection handle for an existing collection. |
| `key (IN)` | A string that contains the document key. Must be `NULL` if inserting a document into a collection configured for auto-generated keys. |
| `keyLength (IN)` | The length of the key string. Must be 0 if inserting a document into a collection configured for auto-generated keys. |
| `content (IN)` | A string containing the JSON content for the document.<br>**Note**: This method works only for JSON documents. |
| `contentLength (IN)` | The length of the document content. |

| Parameter | Description |
|---|---|
| `docFlags (IN)` | The encoding flags used to indicate the encoding of the supplied document's content. Valid flag options are:<br>• `OCI_DEFAULT` — If you use this parameter value then you are, in effect, declaring that the document content is in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle). If that is not the case for a given document, then the result of trying to write the document is unpredictable.<br>• `OCI_SODA_DETECT_JSON_ENC` — Automatically detects the encoding of the document content as either UTF-8, UTF-16 LE, or UTF-16 BE. If you use this parameter value then you are, in effect, declaring that the document content is either UTF-8, UTF-16 LE, or UTF-16 BE. If that is not the case for a given document, then the result of trying to write the document is unpredictable. |
| `errhp (IN/OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if the document insertion into the specified collection is successful.

`OCI_ERROR`, if the document insertion failed. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Inserting Documents into Collections with SODA for C.

## 36.2.27 OCISodaRemove()

Removes the documents matching the attributes set in operation options input handle from the collection. If no attributes are set, then all the documents in the collection are removed.

### Purpose

To remove the documents matching the attributes set in operation options input handle from the collection. If no attributes are set, then all the documents in the collection are removed.

### Syntax

```
sword OCISodaRemove(OCISvcCtx                      *svchp,
                    const OCISodaColl              *coll,
                    const OCISodaOperationOptions   *optns,
                    ub8                            *removeCount,
                    OCIError                       *errhp,
                    ub4                             mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The collection handle for an existing collection. |
| `optns (IN)` | The operation options handle with attributes to drive the remove operation. If no attributes are set, all the documents in the collection are removed. If `OCI_ATTR_SODA_SKIP` and `OCI_ATTR_SODA_LIMIT` attributes are set on the operation options handle, then they are ignored. That is, they do not apply to write operations such as removes. |
| `removeCount (OUT)` | The number of documents removed. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if one or more documents are removed from the specified collection.

`OCI_ERROR`, if the document removal failed. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Removing Documents from a Collection with SODA for C

## 36.2.28 OCISodaRemoveOneWithKey()

Removes a document from a collection given a key.

### Purpose

Remove a document from a collection given a key.

> **Note:**
>
> This is a convenience method for removing a document with a single key. A more general method is `OCISodaRemove`.

### Syntax

```
sword OCISodaRemoveOneWithKey(OCISvcCtx          *svchp,
                              const OCISodaColl  *coll,
                              const OraText      *key,
                              ub4                 keylength,
                              boolean            *isRemoved,
                              OCIError           *errhp,
                              ub4                 mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `coll (IN)` | The collection handle for an existing collection. |
| `key (IN)` | The key of the document to be removed. |
| `keylength (IN)` | The length of key of the document to be removed. |
| `isRemoved (OUT)` | Returns TRUE if a document was found with the specified key and removed; FALSE if no document could be found with the specified key. |

| Parameter | Description |
|---|---|
| `errhp (IN/OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if the document with the specified key was not found. `OCI_SUCCESS` is still returned (and `isRemoved` will be set to `FALSE`). Check that the `isRemoved` parameter is set to `TRUE` to know whether the document has been removed.

`OCI_ERROR`, if the document removal operation failed during execution due to a SQL error. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Removing Documents from a Collection with SODA for C.

## 36.2.29 OCISodaReplOne()

Replaces a document matching the attributes set in operation options input handle.

### Purpose

To replace a document matching the attributes set in operation options input handle. In order to uniquely identify the document to be replaced, at least `OCI_ATTR_SODA_KEY` key must be set in operation options. Other attributes can also be set.

### Syntax

```
sword OCISodaReplOne(OCISvcCtx                        *svchp,
                     const OCISodaColl                *coll,
                     const OCISodaOperationOptions    *optns,
                     OCISodaDoc                       *document,
                     boolean                          *isReplaced,
                  OCIError                       *errhp,
                  ub4                             mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The collection handle for an existing collection. |
| `optns (IN)` | The operation options handle with attributes to drive the replace operation. At least the key, `OCI_ATTR_SODA_KEY` attribute must be set to uniquely identify the document to be replaced. Otherwise the operation returns an error. Other attributes such as `OCI_ATTR_SODA_VERSION`, and `OCI_ATTR_SODA_FILTER` can also be set. If `OCI_ATTR_SODA_SKIP` and `OCI_ATTR_SODA_LIMIT` attributes are set on the operation options handle, then they are ignored. That is, they are not applied to the write operations such as replace. |
| `document (IN)` | The document handle to use as the replacement of the document currently in the collection. Only the content and media type component from this handle is used during the replace operation (that is, this operation replaces the content and media type of the document in the collection). Other components, if set on the input document, are ignored. |
| `isReplaced (OUT)` | Whether the document is replaced or not. Returns `TRUE` if replaced, otherwise it returns `FALSE`. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` - Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` - When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, document is replaced in the specified collection.

`OCI_ERROR`, if the document replacment failed. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Replacing Documents in a Collection with SODA for C.

## 36.2.30 OCISodaReplOneAndGet()

Replaces a document matching the attributes set in operation options input handle, and returns the handle to the result document.

### Purpose

To replace a document matching the attributes set in operation options input handle, and returns the handle to the result document. In order to uniquely identify the document to be replaced, at least `OCI_ATTR_SODA_KEY` key must be set in operation options. Other attributes can also be set.

### Syntax

```
sword OCISodaReplOneAndGet(OCISvcCtx                      *svchp,
                           const OCISodaColl              *coll,
                           const OCISodaOperationOptions  *optns,
                           OCISodaDoc                    **document,
                           boolean                        *isReplaced,
                           OCIError                       *errhp,
                           ub4                             mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The collection handle for an existing collection. |
| `optns (IN)` | The operation options handle with attributes to drive the replace operation. At least the key, `OCI_ATTR_SODA_KEY` attribute must be set to uniquely identify the document to be replaced. Otherwise the operation returns an error. Other attributes such as `OCI_ATTR_SODA_VERSION`, and `OCI_ATTR_SODA_FILTER` can also be set. If `OCI_ATTR_SODA_SKIP` and `OCI_ATTR_SODA_LIMIT` attributes are set on the operation options handle, then they are ignored. That is, they are not applied to the write operations such as replace. |

| Parameter | Description |
|---|---|
| `document (IN/OUT)` | The document handle to use as the replacement of the document currently in the collection. Only the content and media type component from this handle is used during the replace operation (that is, this operation replaces the content and media type of the document in the collection). Other components, if set on the input document, are ignored. Returns the result document through this same parameter as it is an `IN/OUT`.<br>**Caution:**<br>Because the result document is returned using the same parameter, you need to save a reference to the input document handle so that you can free it later, along with the result document handle, using `OCIHandleFree()`. If you do not save a reference to the input document handle, it is overwritten with a reference to the result document, and the input document handle will never be properly freed (it's a memory leak). |
| `isReplaced (OUT)` | Whether the document is replaced or not. Returns TRUE if replaced, otherwise it returns `FALSE`. |
| `errhp (OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

**Returns**

`OCI_SUCCESS`, if the document is replaced in the specified collection.

`OCI_ERROR`, if the document replacement failed due to hard errors. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Replacing Documents in a Collection with SODA for C.

## 36.2.31 OCISodaReplOneAndGetWithCtnt()

Replaces the content of the document matching the attributes set in operation options input handle with new content, and returns the handle to the result document.

**Purpose**

To replace the content of the document matching the attributes set in operation options input handle with new content, and returns the handle to the result document containing all components except for the content, including the updated ones, such as the last-modified timestamp and version. In order to uniquely identify the document to be replaced, at least `OCI_ATTR_SODA_KEY` key must be set in operation options. Other attributes can also be set.

**Note:**

This is a convenience method for replacing a document with a JSON document without constructing a document handle for the replacement document. A more general method is `OCISodaReplOneAndGet`. This method is only for JSON documents. That is, the supplied replacement document must be a JSON document and the target document that is being replaced in the collection may or may not be a JSON document.

### Syntax

```
sword OCISodaReplOneAndGetWithCtnt(OCISvcCtx                  *svchp,
                                   const OCISodaColl          *coll,
                                   const OCISodaOperationOptions *optns,
                                   const void                 *content,
                                   ub4
contentLen,
                                   ub4                         docFlags,
                                   OCISodaDoc                **document,
                                   boolean
*isReplaced,
                                   OCIError                   *errhp,
                                   ub4                         mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The collection handle for an existing collection. |
| `optns (IN)` | The operation options handle with attributes to drive the replace operation. At least the key, `OCI_ATTR_SODA_KEY` attribute must be set to uniquely identify the document to be replaced. Otherwise the operation returns an error. Other attributes such as `OCI_ATTR_SODA_VERSION`, and `OCI_ATTR_SODA_FILTER` can also be set. If `OCI_ATTR_SODA_SKIP` and `OCI_ATTR_SODA_LIMIT` attributes are set on the operation options handle, then they are ignored. That is, they are not applied to the write operations such as replace. |
| `content (IN)` | A string containing the JSON content of the document.<br>**Note**: This method works only for JSON documents. |
| `contentLen (IN)` | The length of the document content. |

| Parameter | Description |
|---|---|
| `docFlags (IN)` | The encoding flags used to indicate the encoding of the supplied document's content. Valid flag options are:<br>• `OCI_DEFAULT` — If you use this parameter value then you are, in effect, declaring that the document content is in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle). If that is not the case for a given document, then the result of trying to write the document is unpredictable.<br>• `OCI_SODA_DETECT_JSON_ENC` — Automatically detects the encoding of the document content as either UTF-8, UTF-16 LE, or UTF-16 BE. If you use this parameter value then you are, in effect, declaring that the document content is either UTF-8, UTF-16 LE, or UTF-16 BE. If that is not the case for a given document, then the result of trying to write the document is unpredictable. |
| `document (OUT)` | Result document contains all components except for the content, including the updated ones, such as the last-modified timestamp and version. |
| `isReplaced (OUT)` | Whether the document is replaced or not. Returns `TRUE` if replaced, otherwise it returns `FALSE`. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if the document is replaced in the specified collection.

`OCI_ERROR`, if the document replacement failed. The `OCIError` parameter has the necessary error information.

**Usage Notes**

None.

**Examples**

For examples, see Replacing Documents in a Collection with SODA for C.

## 36.2.32 OCISodaReplOneAndGetWithKey()

Replaces a document in a collection given a key and returns the handle to the result document.

**Purpose**

To replace a document in a collection given a key and return the handle to the result document containing all components except content, including the updated ones, such as the last-modified timestamp and version.

> **Note:**
>
> This is a convenience method for replacing a document by a key. A more general method is `OCISodaReplOneAndGet`.

**Syntax**

```
sword OCISodaReplOneAndGetWithKey(OCISvcCtx         *svchp,
                                  const OCISodaColl *coll,
                                  const OraText     *key,
                                  ub4                keylength,
                                  OCISodaDoc       **document,
                                  boolean           *isReplaced,
                                  OCIError          *errhp,
                                  ub4                mode);
```

**Parameters**

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `coll (IN)` | The collection handle for an existing collection. |
| `key (IN)` | The key used to identify the document. |
| `keylength (IN)` | The length of the key. |

| Parameter | Description |
|---|---|
| `document (IN/OUT)` | The document handle to use as the replacement of the document currently in the collection. Only the content and media type component from this handle is used during the replace operation (that is, this operation replaces the content and media type of the document in the collection). Other components, if set on the input document, are ignored. Returns the result document through this same parameter as it is an `IN/OUT`.<br>**Caution:**<br>Because the result document is returned using the same parameter, you need to save a reference to the input document handle so that you can free it later, along with the result document handle, using `OCIHandleFree()`. If you do not save a reference to the input document handle, it is overwritten with a reference to the result document, and the input document handle will never be properly freed (it's a memory leak). |
| `isReplaced (OUT)` | Returns `TRUE` if a document was found with the specified key and replaced; `FALSE` if no document could be found with the specified key. |
| `errhp (IN/OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if the document with the specified key was not found. `OCI_SUCCESS` is still returned (and `isReplaced` will be set to `FALSE`). Check that the `isReplaced` parameter is set to `TRUE` to know whether the document has been replaced.

`OCI_ERROR`, if the document replacement operation failed during execution due to a SQL error. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Replacing Documents in a Collection with SODA for C

## 36.2.33 OCISodaReplOneWithCtnt()

Replaces the content of the document matching the attributes set in operation options input handle with new content.

### Purpose

To replace the content of the document matching the attributes set in operation options input handle with new content. In order to uniquely identify the document to be replaced, at least `OCI_ATTR_SODA_KEY` key must be set in operation options. Other attributes can also be set.

**Note:**

This is a convenience method for replacing a document with a JSON document without constructing a document handle for the replacement document. A more general method is `OCISodaReplOne`. This method is only for JSON documents. That is, the supplied replacement document must be a JSON document and the target document that is being replaced in the collection may or may not be a JSON document.

### Syntax

```
sword OCISodaReplOneWithCtnt(OCISvcCtx                      *svchp,
                             const OCISodaColl              *coll,
                             const OCISodaOperationOptions   *optns,
                             const void                     *content,
                             ub4
contentLength,
                             ub4                             docFlags,
                             boolean                        *isReplaced,
                             OCIError                       *errhp,
                             ub4                             mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context. |
| `coll (IN)` | The collection handle for an existing collection. |
| `optns (IN)` | The operation options handle with attributes to drive the replace operation. At least the key, `OCI_ATTR_SODA_KEY` attribute must be set to uniquely identify the document to be replaced. Otherwise the operation returns an error. Other attributes such as `OCI_ATTR_SODA_VERSION`, and `OCI_ATTR_SODA_FILTER` can also be set. If `OCI_ATTR_SODA_SKIP` and `OCI_ATTR_SODA_LIMIT` attributes are set on the operation options handle, then they are ignored. That is, they are not applied to the write operations such as replace. |
| `content (IN)` | A string containing the JSON content of the document.<br>**Note**: This method works only for JSON documents. |
| `contentLength (IN)` | The length of the document content. |

| Parameter | Description |
|---|---|
| `docFlags (IN)` | The encoding flags used to indicate the encoding of the supplied document's content. Valid flag options are:<br>• `OCI_DEFAULT` — If you use this parameter value then you are, in effect, declaring that the document content is in the character set defined by the environment handle (or environment variable `NLS_LANG`, if not set for the handle). If that is not the case for a given document, then the result of trying to write the document is unpredictable.<br>• `OCI_SODA_DETECT_JSON_ENC` — Automatically detects the encoding of the document content as either UTF-8, UTF-16 LE, or UTF-16 BE. If you use this parameter value then you are, in effect, declaring that the document content is either UTF-8, UTF-16 LE, or UTF-16 BE. If that is not the case for a given document, then the result of trying to write the document is unpredictable. |
| `isReplaced (OUT)` | Whether the document is replaced or not. Returns `TRUE` if replaced, otherwise it returns `FALSE`. |
| `errhp (OUT)` | The error handle. |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if the document is replaced in the specified collection.

`OCI_ERROR`, if the document replacement failed. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

**Examples**

For examples, see Replacing Documents in a Collection with SODA for C.

## 36.2.34 OCISodaReplOneWithKey()

Replaces a document in a collection given a key.

**Purpose**

To replace a document in a collection given a key.

> **Note:**
>
> This is a convenience method for replacing a document by a key. A more general method is `OCISodaReplOne`.

**Syntax**

```
sword OCISodaReplOneWithKey (OCISvcCtx          *svchp,
                             const OCISodaColl   *coll,
                             const OraText       *key,
                             ub4                  keylength,
                             OCISodaDoc          *document,
                             boolean             *isReplaced,
                             OCIError            *errhp,
                             ub4                  mode);
```

**Parameters**

| Parameter | Description |
|---|---|
| `svchp (IN)` | The service context handle. |
| `coll (IN)` | The collection handle for an existing collection. |
| `key (IN)` | The key used to identify the document. |
| `keylength (IN)` | The length of the key. |
| `document (IN)` | The document handle to use as the replacement of the document currently in the collection. Only the content and media type component from this handle is used during the replace operation (that is, this operation replaces the content and media type of the document in the collection). Other components, if set on the input document, are ignored. |
| `isReplaced (OUT)` | Returns `TRUE` if a document was found with the specified key and replaced; `FALSE` if no document could be found with the specified key. |
| `errhp (IN/OUT)` | The error handle. |

| Parameter | Description |
|---|---|
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes.<br>• `OCI_SODA_ATOMIC_COMMIT` — When an operation executes in this mode and it completes successfully, then the current transaction is committed after completion. This commits all outstanding changes, not just changes made by the SODA operation. If the operation fails, then only changes made for the SODA operation are rolled back; any uncommitted changes made prior to invocation of the SODA operation are not rolled back. |

### Returns

`OCI_SUCCESS`, if the document with the specified key was not found. `OCI_SUCCESS` is still returned (and `isReplaced` will be set to `FALSE`). Check that the `isReplaced` parameter is set to `TRUE` to know whether the document has been replaced.

`OCI_ERROR`, if the document replacement operation failed during execution due to a SQL error. The `OCIError` parameter has the necessary error information.

### Usage Notes

None.

### Examples

For examples, see Replacing Documents in a Collection with SODA for C

## 36.2.35 OCISodaOperKeysSet()

Sets an array of keys on Operation Options Handle.

### Purpose

Sets an array of keys on Operation Options Handle. Attributes of the operation options handle are set with standard `OCIAttrSet` function. The `OCIAttrSet` function is not used to set the array of keys, because both the array of keys and the array of key lengths need to be set.

> **Note:**
>
> If you use function `OCIAttrSet()` to set attribute `OCI_ATTR_SODA_KEY` on an operation-options handle, and you also use function `OCISodaOperKeysSet()` to set multiple keys on the same handle, then only the latest of the two settings takes effect. The effect of the first function invoked is overridden by the effect of the second.

### Syntax

```
sword OCISodaOperKeysSet (const OCISodaOperationOptions  *operhp,
                          OraText                        **keysArray,
                          ub4                            *lengthsArray,
                          ub4                             count,
                          OCIError                       *errhp,
                          ub4                             mode);
```

### Parameters

| Parameter | Description |
|---|---|
| `operhp (IN)` | Operation options handle |
| `keysArray (IN)` | Array of `oratext*` keys |
| `lengthsArray (IN)` | Array of ub4 lengths corresponding to each key |
| `count  (IN)` | Number of elements in the keys array |
| `errhp (OUT)` | Error handle |
| `mode (IN)` | Specifies the mode of execution. Valid modes are:<br>• `OCI_DEFAULT` — Is the default mode. It means execute the operation as is with no special modes. |

### Returns

`OCI_SUCCESS`, if the keys were set successfully.

### Usage Notes

None.

### Examples

For examples, see Finding Multiple Documents with Specified Document Keys and Removing Documents from a Collection Using Document Keys.

# A

# Handle and Descriptor Attributes

This appendix describes the attributes for OCI handles and descriptors.

OCI handles and descriptors can be read with `OCIAttrGet()`, and modified with `OCIAttrSet()`.

This appendix contains these topics:

- Conventions
- Environment Handle Attributes
- Error Handle Attributes
- Service Context Handle Attributes
- Server Handle Attributes
  - Authentication Information Handle Attributes
  - User Session Handle Attributes
- Administration Handle Attributes
- Connection Pool Handle Attributes
- Session Pool Handle Attributes
- Transaction Handle Attributes
- Statement Handle Attributes
- Bind Handle Attributes
- Define Handle Attributes
- Describe Handle Attributes
- Parameter Descriptor Attributes
- Shard Instance Descriptor Attributes
- SODA Document Handle Attributes
- SODA Collection Handle Attributes
- SODA Output Options Handle Attributes
- SODA Operation Options Handle Attributes
- LOB Descriptor and LOB Locator Attributes
- Complex Object Attributes
  - Complex Object Retrieval Handle Attributes
  - Complex Object Retrieval Descriptor Attributes
- Database Advanced Queuing Descriptor Attributes
  - OCIAQEnqOptions Descriptor Attributes
  - OCIAQDeqOptions Descriptor Attributes

  - OCIAQMsgProperties Descriptor Attributes
  - OCIAQAgent Descriptor Attributes
  - OCIServerDNs Descriptor Attributes
- Subscription Handle Attributes
  - Continuous Query Notification Attributes
  - Continuous Query Notification Descriptor Attributes
  - Notification Descriptor Attributes
  - Invalidated Query Attributes
- Direct Path Loading Handle Attributes
  - Direct Path Context Handle (OCIDirPathCtx) Attributes
  - Direct Path Function Context Handle (OCIDirPathFuncCtx) Attributes
  - Direct Path Function Column Array Handle (OCIDirPathColArray) Attributes
  - Direct Path Stream Handle (OCIDirPathStream) Attributes
  - Direct Path Column Parameter Attributes
- Process Handle Attributes
- Event Handle Attributes

## A.1 Conventions

For each handle type, the attributes that 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 Data Type**
This is the data type of the attribute. If necessary, a distinction is made between the data type 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.

# A.2 Environment Handle Attributes

Lists and describes environment handle attributes.

The following attributes are used for the environment handle.

### 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 Data Type**
`OCIDuration */OCIDuration`

### OCI_ATTR_BIND_DN

**Mode**
READ/WRITE

**Description**
The login name (DN) to use when connecting to the LDAP server.

**Attribute Data Type**
`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, which can considerably improve performance. An attribute value of `TRUE` should only be used when the order in which the objects are flushed is not important. When the attribute value is set to `TRUE`, it is not guaranteed that the order in which the objects are marked dirty is preserved.

> **See Also:**
>
> Object Cache Parameters and About Flushing Changes to the Server

**Attribute Data Type**
`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

**Attribute Data Type**
`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 8 megabytes (MB). Setting this attribute to 0 results in a value of 8 MB being used.

> **See Also:**
>
> Object Cache Parameters

**Attribute Data Type**
`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. When character set ID is UTF-16, an attempt to get this attribute is invalid.

**Attribute Data Type**
`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. When character set ID is UTF-16, an attempt to get this attribute is invalid.

**Attribute Data Type**
`ub2 *`

### OCI_ATTR_ENV_NLS_LANGUAGE

**Mode**
READ/WRITE

**Description**
The name of the language used for the database sessions created from the current environment handle. While getting this value, users should pass an allocated buffer, which will be filled with the language name.

**Attribute Data Type**
`oratext **` or `oratext *`

### OCI_ATTR_ENV_NLS_TERRITORY

**Mode**
READ/WRITE

**Description**
The name of the territory used for the database sessions created from the current environment handle. While getting this value, users should pass an allocated buffer, which will be filled with the territory name.

**Attribute Data Type**
`oratext **` or `oratext *`

### OCI_ATTR_ENV_UTF16

**Mode**
READ

**Description**
Encoding method is UTF-16. The value 1 means that the environment handle is created when the encoding method is UTF-16, whereas 0 means that it is not. This attribute value can only be set by the call to `OCIEnvCreate()` and cannot be changed later.

**Attribute Data Type**
`ub1 *`

### OCI_ATTR_EVTCBK

**Mode**
WRITE

**Description**
This attribute registers an event callback function.

> **See Also:**
>
> HA Event Notification

**Attribute Data Type**
`OCIEventCallback`

### OCI_ATTR_EVTCTX

**Mode**
WRITE

**Description**
This attribute registers a context passed to an event callback.

> **See Also:**
>
> HA Event Notification

**Attribute Data Type**
void *

### 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 Data Type**
`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 is determined at run time.

**Attribute Data Type**
`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 Data Type**
`oratext **/oratext *`

### OCI_ATTR_LDAP_CTX

**Mode**
READ/WRITE

**Description**
The administrative context of the client. This is usually the root of the Oracle Database LDAP schema in the LDAP server.

**Attribute Data Type**
`oratext **/oratext *`

### OCI_ATTR_LDAP_HOST

**Mode**
READ/WRITE

**Description**
The name of the host on which the LDAP server runs.

**Attribute Data Type**
`oratext **/oratext *`

### OCI_ATTR_LDAP_PORT

**Mode**
READ/WRITE

**Description**
The port on which the LDAP server is listening.

**Attribute Data Type**
`ub2 */ub2`

### OCI_ATTR_OBJECT

**Mode**
READ

**Description**
Returns `TRUE` if the environment was initialized in object mode.

**Attribute Data Type**
`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`, and `OCIObjectPin()` is called with the *`pin_option`* parameter set to `OCI_PIN_DEFAULT`, the object is pinned in `OCI_PIN_RECENT` mode.

**Attribute Data Type**
`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:**
>
> About Creating Objects

**Attribute Data Type**
`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 that has been modified in the server by another committed transaction.

> **See Also:**
>
> About Implementing Optimistic Locking

**Attribute Data Type**
`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 Data Type**
`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((void *)envhp, (ub4)OCI_HTYPE_ENV,
           (void *) &heapsz, (ub4 *) 0,
           (ub4)OCI_ATTR_SHARED_HEAPALLOC, errhp);
```

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_SUBSCR_PORTNO

**Mode**
READ/WRITE

**Description**
The client port used to receive notifications. It is set on the client's environment handle.

**Attribute Data Type**
`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 Data Type**
`oratext **/oratext *`

## A.3 Error Handle Attributes

Lists and describes error handle attributes.

The following attributes are used for the error handle.

**OCI_ATTR_DML_ROW_OFFSET**

**Mode**
READ

**Description**
Returns the offset (into the DML array) at which the error occurred.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_ERROR_IS_RECOVERABLE**

**Mode**
READ

**Description**
This attribute is set to `TRUE` if the error in the error handle is recoverable. If the error is not recoverable, it is set to `FALSE`.

**Attribute Data Type**
`Boolean *`

## A.4 Service Context Handle Attributes

Lists and describes service context handle attributes.

The following attributes are used for service context handle.

**OCI_ATTR_CALL_TIMEOUT**

**Mode**
READ/WRITE

**Description**
A database round-trip call using the service context times out within the specified time in milli-seconds, if not completed. When the call times out, a network timeout error is returned. This attribute can be set dynamically. Setting this value stays effective for all subsequent round-trip calls executed using the same service context until a different value is set. To remove the timeout, the value must be set to 0.
This is a client-only change. So, for using this feature, you must use a release 18c client. This feature is database version agnostic, that is, you can use this feature with any supported version of the database.
Timeout values set in different places are effective according to the following order of precedence (where 1 is the highest precedence):

1. Call timeout set on the OCI handle using `OCIAttrSet()`.

2. Timeout set with the OCI attributes, `OCI_ATTR_SEND_TIMEOUT` and `OCI_ATTR_RECEIVE_TIMEOUT`.
3. Timeout set in the `sqlnet.ora` file with the parameters, `SQLNET.RECV_TIMEOUT` and `SQLNET.SEND_TIMEOUT`.

Error returned to the application:

```
ORA-03136 inbound connection timed out
```

The Call timeout is on each individual round-trip between OCI and Oracle Database. Each OCI method or operation may require zero or more round-trips to Oracle Database. The timeout value applies to each round-trip individually, not to the sum of all round-trips. Time spent processing in OCI before or after the completion of each round-trip is not counted.

- If the time from the start of any round-trip to the completion of the same round-trip exceeds the timeout value, then the operation is halted and an Oracle error is returned.
- In the case where an OCI operation requires more than one round-trip and each round-trip takes less than the timeout value, then no timeout occurs, even if the sum of all round-trip calls exceeds the timeout value.
- If no round-trip is required, the operation will never be interrupted.

After a timeout occurs, OCI attempts to clean up the internal connection state. The cleanup is allowed to take another timeout value. If the cleanup was successful, an `ORA-3136` will be returned and the application can continue to use the connection. For small values of the timeout value, the connection cleanup may not complete successfully within the additional timeout period. In this case an `ORA-3114` is returned and the connection will no longer be usable. It should be released. Alternatively, the `OCI_ATTR_BREAK_ON_NET_TIMEOUT` attribute can additionally be set to `FALSE`. This setting drops the connection, eliminating calls to `OCIBreak()` and `OCIReset()` when a timeout occurs.

#### Attribute Data Type

`ub4*/ub4`

#### Example

```
static void func1( )
{
sword status = 0;
  OCIStmt *stmthp = (OCIStmt *)0;
  OraText *sqlStmt = (OraText *)"SELECT EMPNO FROM SCOTT.EMP";
  ub4 set_timeout= 200; /* in milliseconds */
  ub4 get_timeout = 0;

  Checkerr (errhp, \
      OCIStmtPrepare2 ((OCISvcCtx *)svchp, (OCIStmt **)&stmthp, \
      (OCIError *)errhp, (OraText *)sqlStmt, (ub4)strlen((char *)sqlStmt),
\
      (OraText *)NULL, (ub4) 0, (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT), \
      (OraText *)"OCIStmtPrepare2");
  /* Set the call Timeout (in milliseconds) */
    Checkerr (errhp,
    OCIAttrSet(svchp, (ub4) OCI_HTYPE_SVCCTX,
     (dvoid *) &set_timeout, (ub4) 0,
     (ub4) OCI_ATTR_CALL_TIMEOUT, errhp),
     "OCIAttrSet OCI_ATTR_CALL_TIMEOUT");

    if ((status = OCIStmtExecute ((OCISvcCtx *)svchp, \
      (OCIStmt *)stmthp, (OCIError *)errhp, (ub4)1, (ub4)0, \
      (OCISnapshot *)0, (OCISnapshot *)0, (ub4)OCI_DEFAULT)) \
      != OCI_SUCCESS)
    {
      printf ( "OCIStmtExecute Failed with timeout: %dms\n", set_timeout);
      Checkerr (errhp, status,(OraText *)"OCIStmtExecute");
    }

  /* Get the call Timeout (in milliseconds) */
    Checkerr (errhp,
    OCIAttrGet(svchp, (ub4) OCI_HTYPE_SVCCTX,
     (dvoid *) &get_timeout, (ub4) 0,
     (ub4) OCI_ATTR_CALL_TIMEOUT, errhp),
     "OCIAttrGet OCI_ATTR_CALL_TIMEOUT");

  Checkerr (errhp, \
      OCIStmtRelease ((OCIStmt *)stmthp, (OCIError *)errhp,(dvoid *)NULL, \
      0, OCI_DEFAULT), (oratext *)"StmtRelease");
} /* End of func1 */
```

## OCI_ATTR_ENV

**Mode**
READ

**Description**
Returns the environment context associated with the service context.

**Attribute Data Type**
`OCIEnv **`

## OCI_ATTR_INSTNAME

See the `OCI_ATTR_INSTNAME` attribute in Shard Instance Descriptor Attributes for more information about its use on the service context.

## 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 Data Type

`ub1 *`

#### Example

The following code sample shows how this attribute is used:

```
in_v8_mode = 0;
OCIAttrGet ((void *)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_MAX_CHARSET_RATIO

#### Mode

READ

#### Description

Returns the maximum character set expansion ratio from server to client character set. Using these attributes is useful in scenarios where there are different character sets between server and client. This is useful to allocate optimal memory of buffers before conversion so that when data is returned from the database, sufficient space can be allocated to hold it.

#### Attribute Data Type

ub4 *

#### Example

```
size_t   cratio;
OCIAttrGet((void *)svchp, (ub4)OCI_HTYPE_SVCCTX, (size_t *)&cratio, (ub4)
0, OCI_ATTR_MAX_CHARSET_RATIO, errhp);
printf("Conversion ratio from server to client character set is %d\n",
cratio);
```

### OCI_ATTR_MAX_NCHARSET_RATIO

#### Mode

READ

#### Description

Returns the maximum character set expansion ratio from server to client national character set. Using these attributes is useful in scenarios where there are different national character sets between server and client. This is useful to allocate optimal memory of buffers before conversion so that when data is returned from the database, sufficient space can be allocated to hold it.

#### Attribute Data Type

ub4 *

**Example**

```
size_t   cratio;
OCIAttrGet((void *)svchp, (ub4)OCI_HTYPE_SVCCTX, (size_t *)&cratio, (ub4)
0, OCI_ATTR_MAX_NCHARSET_RATIO, errhp);
printf("Conversion ratio from server to client ncharset is %d\n", cratio);
```

### 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 Data Type**
`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 Data Type**
`OCISession **`/ `OCISession *`

### OCI_ATTR_STMTCACHE_CBK

**Mode**
READ/WRITE

**Description**
Used to get and set the application's callback function on the `OCISvcCtx` handle. This function, if registered on `OCISvcCtx`, is called when a statement in the statement cache belonging to this service context is purged or when the session is ended.
The callback function must be of this prototype:

```
sword (*OCICallbackStmtCache)(void *ctx, OCIStmt *stmt, ub4 mode)
```

`ctx`: IN argument. This is the same as the context the application has set on the current statement handle.
`stmt`: IN argument. This is the statement handle that is being purged from the cache.
`mode`: IN argument. This is the mode in which the callback function is being called. Currently only one value is supported, `OCI_CBK_STMTCACHE_STMTPURGE`, which means the callback is being called during purging of the current statement.

**Attribute Data Type**

```
sword (*OCICallbackStmtCache)(void *ctx, OCIStmt *stmt, ub4 mode)
```

### 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 nonpooled. Statement caching can be enabled by setting the attribute to a nonzero size and disabled by setting it to zero.

**Attribute Data Type**
`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 Data Type**
`OCITrans **` / `OCITrans *`

### OCI_ATTR_VARTYPE_MAXLEN_COMPAT

**Mode**
READ

**Description**
Returns `OCI_ATTR_MAXLEN_COMPAT_EXTENDED` if the `init.ora` parameter `max_string_size = extended` or returns `OCI_ATTR_MAXLEN_COMPAT_STANDARD` if the `init.ora` parameter `max_string_size = standard`.

**Attribute Data Type**
`ub1 *`

## A.5 Server Handle Attributes

Lists and describes server handle attributes.

The following attributes are used for the server handle.

**See Also:**

The following event handle attributes are also available for the server handle:

- OCI_ATTR_DBDOMAIN
- OCI_ATTR_DBNAME
- OCI_ATTR_INSTNAME
- OCI_ATTR_INSTSTARTTIME
- OCI_ATTR_SERVICENAME

OCIXStreamOutLCRReceive() for more information about using the following attributes

### OCI_ATTR_ACCESS_BANNER

**Mode**
READ

**Description**
Displays an unauthorized access banner from a file.

**Attribute Data Type**
oratext **

### OCI_ATTR_BREAK_ON_NET_TIMEOUT

**Mode**
READ/WRITE

**Description**
This attribute determines whether OCI sends a break after a network time out or not.

**Attribute Data Type**
ub1 *

### OCI_ATTR_ENV

**Mode**
READ

**Description**
Returns the environment context associated with the server context.

**Attribute Data Type**
`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 for distributed transaction coordination. Server database names can only be accessed only if the database is open at the time the `OCISessionBegin()` call is issued.

**Attribute Data Type**
`oratext **`/ `oratext *`

### OCI_ATTR_FOCBK

**Mode**
READ/WRITE

**Description**
Sets the failover callback to the callback definition structure of type `OCIFocbkStruct` as part of failover callback registration and unregistration on the server context handle.

> **See Also:**
>
> Transparent Application Failover in OCI

**Attribute Data Type**
`OCIFocbkStruct *`

### OCI_ATTR_INTERNAL_NAME

**Mode**
READ/WRITE

**Description**
Sets the client database name that is recorded when performing global transactions. The DBA can use the name to track transactions that may be pending in a prepared state due to failures.

**Attribute Data Type**
`oratext **` / `oratext *`

### OCI_ATTR_INSTNAME

See the `OCI_ATTR_INSTNAME` attribute in Shard Instance Descriptor Attributes for more information about its use as a server handle attribute.

### 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 Data Type**
`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 is returned.

**Attribute Data Type**
ub1 */ub1

> **See Also:**
>
> Nonblocking Mode in OCI

### OCI_ATTR_RECEIVE_TIMEOUT

**Mode**
READ/WRITE

**Description**
Time specified in milliseconds that a client waits for response data from the database server. When set to a value of zero, the timeout functionality is disabled and the client may wait a long period of time for a response from a database server.
The semantics of this attribute is the same as that of `SQLNET.RECV_TIMEOUT` that can be specified in the `sqlnet.ora` file. When set on the server handle, the value overrides the one specified in the `sqlnet.ora` file.

**Attribute Data Type**
`ub4*/ub4`

### OCI_ATTR_SEND_TIMEOUT

**Mode**
READ/WRITE

**Description**
Time specified in milliseconds that a client waits to complete send operations to the database server. When set to a value of zero, the timeout functionality is disabled and the client may wait a long period of time to complete sending the data to the database server.

The semantics of this attribute is the same as that of `SQLNET.SEND_TIMEOUT` that can be specified in the `sqlnet.ora` file. When set on the server handle, the value overrides the one specified in the `sqlnet.ora` file.

**Attribute Data Type**
`ub4*/ub4`

### OCI_ATTR_SERVER_GROUP

**Mode**
READ/WRITE

**Description**
An alphanumeric string not exceeding 30 characters specifying the server group. This attribute can only be set *after* calling `OCIServerAttach()`.

> **See Also:**
>
> Password and Session Management

**Attribute Data Type**
`oratext **/oratext *`

### OCI_ATTR_SERVER_STATUS

**Mode**
READ

**Description**
Returns the current status of the connection by doing a light weight connection health check. Values are:

- `OCI_SERVER_NORMAL` - The connection is alive.
- `OCI_SERVER_NOT_CONNECTED` - The connection has been terminated.

**Attribute Data Type**
ub4 *

**Example**
The following code sample shows how this parameter is used:

```
ub4 serverStatus = 0
OCIAttrGet((void *)srvhp, OCI_HTYPE_SERVER,
        (void *)&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

**Attribute Data Type**
boolean *

**OCI_ATTR_USER_MEMORY**

**Mode**
READ

**Description**
If the handle was allocated with extra memory, this attribute returns a pointer to the user memory. A `NULL` pointer is returned for those handles not allocated with extra memory.

> **See Also:**
>
> Custom Pooling: Tagged Server Handles

**Attribute Data Type**
void *

## A.5.1 Authentication Information Handle Attributes

Lists and describes authentication information handle attributes.

These attributes also apply to the user session handle.

> **See Also:**
>
> User Session Handle Attributes

**OCI_ATTR_FIXUP_CALLBACK**

**Mode**
READ/WRITE

**Description**
Specifies on the authentication handle attribute the callback passed to `OCISessionGet()` for applications not using an OCISessionPool or using custom pools. For applications using OCISessionPool, this attribute must be set on the authentication handle, which in turn must be set on the session pool handle as the attribute `OCI_ATTR_SPOOL_AUTH`.

**See Also:**

- Session Pool Handle Attributes for more information
- PL/SQL Callback for Session State Fix Up for more information

**Attribute Data Type**
`oratext *` to be provided in the format `schma.package.callback_function`

## A.5.2 User Session Handle Attributes

Lists and describes user session handle attributes.

The following user session handle 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 Data Type**
`oratext *`

**Example**

```
OCIAttrSet(session, OCI_HTYPE_SESSION,(void *)"insert into employees",
          (ub4)strlen("insert into employees"), OCI_ATTR_ACTION, error_handle);
```

**OCI_ATTR_APPCTX_ATTR**

**Note:**

This attribute is not supported with database resident connection pooling.

**Mode**
WRITE

**Description**
Specifies an attribute name of the externally initialized context.

**See Also:**

Session Handle Attributes Used to Set an Externally Initialized Context

**Attribute Data Type**
`oratext *`

### OCI_ATTR_APPCTX_LIST

**Note:**

This attribute is not supported with database resident connection pooling.

**Mode**
READ

**Description**
Gets the application context list descriptor for the session.

**Attribute Data Type**
`OCIParam **`

### OCI_ATTR_APPCTX_NAME

**Note:**

This attribute is not supported with database resident connection pooling.

**Mode**
WRITE

**See Also:**

Session Handle Attributes Used to Set an Externally Initialized Context

**Description**
Specifies the namespace of the externally initialized context.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_APPCTX_SIZE

> **Note:**
>
> This attribute is not supported with database resident connection pooling.

**Mode**
WRITE

**Description**
Initializes the externally initialized context array size with the number of attributes.

**Attribute Data Type**
`ub4`

### OCI_ATTR_APPCTX_VALUE

> **Note:**
>
> This attribute is not supported with database resident connection pooling.

**Mode**
WRITE

**Description**
Specifies a value of the externally initialized context.

**Attribute Data Type**
`oratext *`

> **See Also:**
>
> Session Handle Attributes Used to Set an Externally Initialized Context

### OCI_ATTR_AUDIT_BANNER

**Mode**
READ

**Description**
Displays a user actions auditing banner from a file.

**Attribute Data Type**
oratext **

### OCI_ATTR_AUTOTUNING_ENABLED

**Mode**
READ

**Description**
Checks if the client session has OCI auto tuning enabled on OCI client statement caching.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_CALL_TIME

**Mode**
READ

**Description**
Returns the server-side time for the preceding call in microseconds.

**Attribute Data Type**
`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 Data Type**
`ub1 *`

### OCI_ATTR_CLIENT_IDENTIFIER

**Mode**
WRITE

**Description**
Specifies the user identifier in the session handle. Can be up to 64 bytes long. The value is automatically truncated if the supplied input is more than 64 bytes. It can contain the user name, but do not include the password for security reasons. The first character of the identifier should not be ':'. If it is, the behavior is unspecified.

**Attribute Data Type**
`oratext *`

**Example**

```
OCIAttrSet(session, OCI_HTYPE_SESSION,(void *)"janedoe",
           (ub4)strlen("janedoe"), OCI_ATTR_CLIENT_IDENTIFIER,
           error_handle);
```

## OCI_ATTR_CLIENT_INFO

**Mode**
WRITE

**Description**
Stores additional client application information. Can also be set by the `DBMS_APPLICATION_INFO` package. It is stored in the `V$SESSION` view. Can be up to 64 bytes long.

**Attribute Data Type**
`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 Data Type**
boolean */boolean

## OCI_ATTR_CONNECTION_CLASS

**Mode**
READ/WRITE

**Description**
This attribute of `OCIAuthInfo` handle explicitly names the connection class (a string of up to 128 characters) for a database resident connection pool.

**Attribute Data Type**
oratext **/oratext *

## 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 that you then enter.
When a client using this attribute communicates with a server that has a software release earlier than Oracle Database 10*g* Release 2, the `OCIAttrSet()` call is ignored. This attribute is also readable by `OCIAttrGet()`.

**Attribute Data Type**
`oratext */oratext *`

**Example**

```
text schema[] = "hr";
err = OCIAttrSet( (void ) mysessp, OCI_HTYPE_SESSION, (void *)schema,
      (ub4)strlen( (char *)schema), OCI_ATTR_CURRENT_SCHEMA, (OCIError *)myerrhp);
```

### OCI_ATTR_DBOP

**Mode**
Write

**Description**
The name of the database operation set by the client application to be monitored in the database. When you want to end monitoring the current running database operation, set the value to `NULL`. Can be up to 30 bytes long.

> **See Also:**
>
> Attributes for End-to-End Application Tracing
> *Oracle Database SQL Tuning Guide*
> *Oracle Database 2 Day + Performance Tuning Guide*

**Attribute Data Type**
`oratext *`

**Example**

```
(void) OCIAttrSet((dvoid *) sess1, (ub4) OCI_HTYPE_SESSION,
                  (dvoid *) dbopname, (ub4) strlen((char *)dbopname),
                  (ub4) OCI_ATTR_DBOP, errhp);
```

### OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE

**Mode**
READ/WRITE

**Description**
Allows the user to enable prefetching for all the LOB locators fetched in the session. Specifies the default prefetch buffer size for each LOB locator.

**Attribute Data Type**
`ub4 */ub4`

### OCI_ATTR_DISTINGUISHED_NAME

**Mode**
WRITE

**Description**
Specifies the distinguished name of the client for use in proxy authentication.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_DRIVER_NAME

**Mode**
READ/WRITE

**Description**
Specifies the name of the driver layer using OCI, such as JDBC, ODBC, PHP, SQL*Plus, and so on. Names starting with "ORA$" are reserved also. A future application can choose its own name and set it as an aid to fault diagnosability. Set this attribute before executing `OCISessionBegin()`. Pass an array containing up to 9 single-byte characters, including the null terminator. This data is not validated and is passed directly to the server to be displayed in a `V$SESSION_CONNECT_INFO` or `GV$SESSION_CONNECT_INFO` view. OCI accepts driver names of up to 30 characters, which are displayed in these views. For older databases, only the first 8 characters are displayed.

**Attribute Data Type**
`oratext **/oratext *`

**Example**

```
...
oratext client_driver[9];
...
checkerr(errhp, OCIAttrSet(authp, OCI_HTYPE_SESSION,
                client_driver, (ub4)(strlen(client_driver)),
                OCI_ATTR_DRIVER_NAME, errhp));

checkerr(errhp, OCISessionBegin(svchp, errhp, authp, OCI_CRED_RDBMS, OCI_DEFAULT);
...
```

### OCI_ATTR_EDITION

**Mode**
READ/WRITE

**Description**
Specifies the edition to be used for this session. If a value for this attribute has not been set explicitly, then the value in the environment variable `ORA_EDITION` is returned.

**Attribute Data Type**
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 an Oracle database on its behalf.

**Attribute Data Type**
`oratext **`.

### OCI_ATTR_LTXID

**Mode**
READ

**Description**
This attribute is defined for the session handle and is used to override the default LTXID (logical transaction ID). Applications that associate the logical session with the web user may want to explicitly attach the LTXID to a physical session and explicitly detach the LTXID when the request is complete.
In `READ` mode, this attribute is used to retrieve the LTXID embedded in the OCISession handle.

**Attribute Data Type**
`ub1 *` (with length; value is copied; is really a `ub1` array)

### OCI_ATTR_MAX_OPEN_CURSORS

**Mode**
READ

**Description**
The maximum number of SQL statements that can be opened in one session. On the server's parameter file, this value is set using the parameter `open_cursors`. The OCI user should leave some threshold and not reach this limit because the server can also open internal statements (cursors) as part of processing user calls. Applications can use this attribute to limit the number of statement handles opened on a given session.
This attribute returns a proper value only when connected to a 12.1 server or later.
If the cursors in the server session exceed the open cursor setting, then the server returns an error to the client saying that the value for max cursors is exceeded.
Also, note that this value should only be looked at from the session handle after an `OCISessionGet()` or equivalent login call has been done.

**Attribute Data Type**
`ub4 *`

**Example**

```
OCIAttrGet((void *)usrhp, OCI_HTYPE_SESSION,
 (void *)&ub4localvariable, (ub4 *)0, OCI_ATTR_MAX_OPEN_CURSORS, errhp);
```

### 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 system or different systems. For a session to be cloned, the session must be authenticated as migratable.

**See Also:**

Password and Session Management

**Attribute Data Type**
`ub1 *`

**Example**
The following code sample shows how this attribute is used:

```
OCIAttrSet ((void *) authp, (ub4)OCI_HTYPE_SESSION, (void *) 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 use `NULL` if there is no new module. Can be up to 48 bytes long.

**Attribute Data Type**
`oratext *`

**Example**

```
OCIAttrSet(session, OCI_HTYPE_SESSION,(void *)"add_employee",
           (ub4)strlen("add_employee"), OCI_ATTR_MODULE, error_handle);
```

### OCI_ATTR_ORA_DEBUG_JDWP

**Mode**
WRITE

**Description**
Specifies the external debugger's location the database session is to connect to after logon. This allows the user to initiate debugging of the database application.
The format of the attribute is a list of parameter names and values separated by semi-colons. The parameters will be passed to the `DBMS_DEBUG_JDWP_CUSTOM.CONNECT_DEBUGGER` package API to connect the database session to the debugger. The parameter name can be a quoted or unquoted PL/SQL identifier. The parameter value is a character string, either quoted by single-quotes, or unquoted, which is terminated by a semi-colon. To escape a single quote in a quoted string, use two consecutive single quotes. No space is allowed anywhere between the parameter names and their values.
The attribute should contain at least the `host` and `port` parameters that specify the Internet Protocol dotted address or host name of the computer the debugger is running, and the TCP port number the debugger is listening to for debug connections.

**Attribute Data Type**
`oratext *`

**Example**

```
OCIAttrSet(session,
            OCI_HTYPE_SESSION,
           (void *)"janedoe",
           (ub4)strlen("janedoe"),
            OCI_ATTR_CLIENT_IDENTIFIER,
            error_handle);
```

### OCI_ATTR_PASSWORD

**Mode**
WRITE

**Description**
Specifies a password to use for authentication.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_PROXY_CLIENT

**Mode**
WRITE

**Description**
Specifies the target user name for access through a proxy.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_PROXY_CREDENTIALS

**Mode**
WRITE

**Description**
Specifies that the credentials of the application server are to be used for proxy authentication.

**Attribute Data Type**
`OCISession`

### OCI_ATTR_PURITY

**Mode**
READ/WRITE

**Description**
An attribute of the `OCIAuthInfo` handle for database resident connection pooling. Values are `OCI_ATTR_PURITY_NEW`, the application requires a session not tainted with any prior session state; or `OCI_ATTR_PURITY_SELF`, the session can have been used before. If the application does not specify the purity when invoking `OCISessionGet()`, then the purity value `OCI_ATTR_PURITY_DEFAULT` is assumed. This later translates to either `OCI_ATTR_PURITY_NEW` or `OCI_ATTR_PURITY_SELF` depending upon whether the application logic is set up to use a new session or to reuse a pooled session.

**Attribute Data Type**
ub4 */ub4

### OCI_ATTR_SESSION_STATE

**Mode**
READ/WRITE

**Description**
Specifies the current state of the database session. Set to `OCI_SESSION_STATEFUL` if the session is required to perform a database task. If the application is no longer dependent on the current session for subsequent database activity, set to `OCI_SESSION_STATELESS`. This attribute is currently applicable only when connected to a Database Resident Connection Pool. It should be used if the application does custom session pooling and does not use `OCISessionPool()`.

**Attribute Data Type**
ub1 */ ub1

### OCI_ATTR_SHARDING_KEY

**Mode**
WRITE

**Description**
For the shard and chunk, specifies the sharding key for the connection request from an OCI session pool or standalone connection.
The `OCIShardingKey *` supplied to `OCIAttrSet()` needs to be created with `OCIDescriptorAlloc()` specifying the sharding descriptor and then using as many calls as is needed to `OCIShardingKeyColumnAdd()` to create the compound sharding key. See OCIShardingKeyColumnAdd() for more information and an example.
For custom pools, this attribute associates a sharding key to a given connection.

**Attribute Data Type**
`OCIShardingKey *`

**Example**

```
/* Allocate the super sharding key descriptor. */
OCIDescriptorAlloc(envhp,(dvoid **)&shardKey,
               OCI_DTYPE_SHARDING_KEY, 0,(dvoid **)0)))
text *name = “KK”;
text *gname = “GOLD”;
int  empid = 150;

/* Add all the columns of the key to form the final shard key. */
OCIShardingKeyColumnAdd(shardKey,(ub1*)&empid, sizeof(empid),
                               SQLT_INT, errhp, OCI_DEFAULT);
OCIShardingKeyColumnAdd(shardKey, name, strlen(name),
                               SQLT_CHAR, errhp, OCI_DEFAULT));

OCIShardingKey *shardKey;
OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
            shardKey, sizeof(shardKey),
            OCI_ATTR_SHARDING_KEY, errhp);
```

### OCI_ATTR_SHARDING_KEY_B64

**Mode**
WRITE

**Description**
Gets the base64 representation of sharding key and super sharding key values for diagnostic purposes.

**Attribute Data Type**
`Oratext *`

**Example**

```
char * skeyValue;
ub4    skeyValueLen;
OCIAttrGet((dvoid *)     OCIShardingKey,
           (ub4)         OCI_DTYPE_SHARDING_KEY,
           (dvoid *)     &skeyValue,
           (ub4 *)       &skeyValueLen,
                         OCI_ATTR_SHARDING_KEY_B64,
           (OCIError * ) errhp);
```

### OCI_ATTR_SUPER_SHARDING_KEY

**Mode**
WRITE

**Description**
For the shard and group of chunks, specifies the super sharding key for the connection request from an OCI session pool or standalone connection.
The `OCIShardingKey *` supplied to `OCIAttrSet()` needs to be created with `OCIDescriptorAlloc()` specifying the sharding descriptor and then using as many calls as is needed to `OCIShardingKeyColumnAdd()` to create the compound shard key. See OCIShardingKeyColumnAdd() for more information and an example.
For custom pools, this attribute associates a shard group key to a given connection.

**Attribute Data Type**
`OCIShardingKey *`

**Example**

```
/* Allocate the sharding key descriptor. */
OCIDescriptorAlloc(envhp,(dvoid **)&shardGroupKey,
                   OCI_DTYPE_SHARDING_KEY, 0, (dvoid **)0));

/* Add the column of the key to form the final super sharding key. */
OCIShardingKeyColumnAdd(shardGroupKey, errhp, gname, strlen(gname),
                     SQLT_CHAR, OCI_DEFAULT));
```

```
OCIShardingKey *shardGroupKey;
OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
            shardGroupKey, sizeof(shardGroupKey),
            OCI_ATTR_SUPER_SHARDING_KEY, errhp));
```

### OCI_ATTR_TRANS_PROFILE_FOREIGN

**Mode**
READ

**Description**
Specifies whether a SQL translation profile for translation of foreign SQL syntax is set in the current session or not.

**Attribute Data Type**
`boolean`

**Example**

```
status = OCIAttrGet(authp,
                    OCI_HTYPE_SESSION,
                    (void *)&foreign_sql_syntax,
                    (ub4 *)NULL,
                    OCI_ATTR_TRANS_PROFILE_FOREIGN,
                    errhp);
```

### OCI_ATTR_TRANSACTION_IN_PROGRESS

**Mode**
READ

**Description**
If `TRUE`, then the referenced session has a currently active transaction.
If `FALSE`, then the referenced session does not have a currently active transaction.

**Attribute Data Type**
`boolean *`

**Example**

```
{  boolean txnInProgress;

  OCIAttrGet(usrhp, OCI_HTYPE_SESSION,
             &txnInProgress, (ub4 *)0,
             OCI_ATTR_TRANSACTION_IN_PROGRESS,
             errhp);
}
```

### OCI_ATTR_USERNAME

**Mode**
READ/WRITE

**Description**
Specifies a user name to use for authentication.

**Attribute Data Type**
`oratext **/oratext *`

## A.6 Administration Handle Attributes

Lists and describes administration handle attributes.

The following attributes are used for the administration handle.

**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 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 Data Type**
`oratext */oratext *`

## A.7 Connection Pool Handle Attributes

Lists and describes connection pool handle attributes.

The following attributes are used for the connection pool handle.

**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 remain active.

**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 Data Type**
`ub4 */ub4`

**OCI_ATTR_CONN_NOWAIT**

**Mode**
READ/WRITE

**Description**

This attribute determines if retrial for a connection must be performed when all connections in the pool are found to be busy and the number of connections has 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 until it gets a connection.

When read, the attribute value is returned as `TRUE` if it has been set.

**Attribute Data Type**

`ub1 */ub1`

### OCI_ATTR_CONN_BUSY_COUNT

**Mode**

READ

**Description**

Returns the number of busy connections.

**Attribute Data Type**

`ub4 *`

### OCI_ATTR_CONN_OPEN_COUNT

**Mode**

READ

**Description**

Returns the number of open connections.

**Attribute Data Type**

`ub4 *`

### OCI_ATTR_CONN_MIN

**Mode**

READ

**Description**

Returns the number of minimum connections.

**Attribute Data Type**

`ub4 *`

### OCI_ATTR_CONN_MAX

**Mode**

READ

**Description**

Returns the number of maximum connections.

**Attribute Data Type**

`ub4 *`

**OCI_ATTR_CONN_INCR**

**Mode**
READ

**Description**
Returns the connection increment parameter.

**Attribute Data Type**
`ub4 *`

## A.7.1 Session Pool Handle Attributes

Lists and describes session pool handle attributes.

The following attributes are used for the session pool handle.

**OCI_ATTR_SPOOL_AUTH**

**Mode**
WRITE

**Description**
To make pre-session creation attributes effective on the sessions being retrieved from the session pool, this attribute can be set on the session pool handle. Currently only the following attributes can be set on this `OCIAuthInfo` handle:
`OCI_ATTR_DRIVER_NAME`
`OCI_ATTR_EDITION`
`OCI_ATTR_FIXUP_CALLBACK`
If any other attributes are set on the `OCIAuthInfo` handle and the `OCIAuthInfo` handle is set on the session pool handle, an error results. Moreover, the `OCIAuthInfo` handle should be set on the session pool handle only before calling `OCISessionPoolCreate()` with the session pool handle. Setting it after `OCISessionPoolCreate()` results in an error.

**Attribute Data Type**
`OCIAuthInfo *`

**OCI_ATTR_SPOOL_BUSY_COUNT**

**Mode**
READ

**Description**
Returns the number of busy sessions checked out from the pool.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_FOCBK**

**Mode**
READ/WRITE

**Description**
On `OCIAttrSet()`, defines a TAF callback and context to be associated with each session in the OCI session pool.
On `OCIAttrGet()`, populates the `OCIFocbkStruct` with the TAF callback and context defined for the session pool.

> **Note:**
>
> - This attribute is already supported for handles of type OCIServer and OCICPool.
> - A TAF callback set on the server handle will override the callback set on the session pool.

**Attribute Data Type**
`OCIFocbkStruct *`

### 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 reached the maximum or the pool must create new connections. Values are:

- `OCI_SPOOL_ATTRVAL_WAIT` - The thread waits and blocks until a session is freed or a new one is created. This is the default value.
- `OCI_SPOOL_ATTRVAL_TIMEDWAIT` - Keep trying internally for a free session until the time out set by `OCI_ATTR_SPOOL_WAIT_TIMEOUT` expires.
- `OCI_SPOOL_ATTRVAL_NOWAIT` - An error is returned if there are no free connections or if the pool must create a new connection.
- `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. In this case, the server returns the following error:

  ```
  ORA 00018 - Maximum number of sessions exceeded
  ```

  In this case, the error is propagated to the session pool user.

  When read, the appropriate attribute value is returned.

**Attribute Data Type**
`ub1 */ ub1`

### OCI_ATTR_SPOOL_INCR

**Mode**
READ

**Description**
Returns the session increment parameter.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_SPOOL_MAX

**Mode**
READ

**Description**
Returns the number of maximum sessions.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_SPOOL_MAX_LIFETIME_SESSION

**Mode**
READ/WRITE

**Description**
This attribute sets the lifetime (in seconds) for all the sessions in the pool. Sessions in the pool are terminated when they have reached their lifetime. In the case when `OCI_ATTR_SPOOL_TIMEOUT` is also set, the session will be terminated if either the idle time out happens or the max lifetime setting is exceeded.

**Attribute Data Type**
ub4 */ ub4

### OCI_ATTR_SPOOL_MAX_USE_SESSION

**Mode**
WRITE

**Description**
Sets the maximum number of times one session can be checked out of the session pool, after which the session is automatically destroyed. The default value is 0, which means there is no limit. This value can also be set in the `oraaccess.xml` client-side configuration file under the `<session_pool>` section as `<max_use_session>`.

**Attribute Data Type**
`ub4`

### OCI_ATTR_SPOOL_MIN

**Mode**
READ

**Description**
Returns the number of minimum sessions.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_SPOOL_OPEN_COUNT

**Mode**
READ

**Description**
Returns the number of open sessions.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_SPOOL_STMTCACHESIZE

**Mode**
READ/WRITE

**Description**
Sets the default statement cache size to this value for each of the sessions in a session pool. 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

**Attribute Data Type**
`ub4 *`/ `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. OCI only checks for timed out sessions when it releases one back to the pool. See OCI_ATTR_SPOOL_MAX_LIFETIME_SESSION for more information.

**Attribute Data Type**
`ub4 */ ub4`

### OCI_ATTR_SPOOL_WAIT_TIMEOUT

**Mode**
READ/WRITE

**Description**
This is the time out (in milliseconds) applied for the `OCISessionGet()` call while waiting for a free session, when the `OCI_ATTR_SPOOL_GETMODE` property is set to `OCI_SPOOL_ATTRVAL_TIMEDWAIT`. This is also applicable in case of sharding, where the `OCISessionGet()` call waits for a free session to an instance with writable chunks.

**Attribute Data Type**
`ub4 */ ub4`

**OCI_ATTR_SPOOL_MAX_PER_SHARD**

**Mode**
READ/WRITE

**Description**
Sets the maximum number of sessions per shard. This attribute is set to ensure that the pool is balanced towards each shard.

**Attribute Data Type**
`ub4 *`

## A.8 Transaction Handle Attributes

Lists and describes transaction handle attributes.

The following attributes are used for the transaction handle.

**OCI_ATTR_TRANS_NAME**

**Mode**
READ/WRITE

**Description**
Can be used to establish or read a text string that 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 Data Type**
`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 Data Type**
`ub4 *` (READ) / `ub4` (WRITE)

**OCI_ATTR_XID**

**Mode**
READ/WRITE

**Description**
Can set or read an XID that identifies a transaction.

**Attribute Data Type**
`XID **` (READ) / `XID *` (WRITE)

## A.9 Statement Handle Attributes

Lists and describes statement handle attributes.

The following attributes are used for the statement handle.

### OCI_ATTR_BIND_COUNT

**Mode**
READ

**Description**
Returns the number of bind positions on the statement handle.

**Attribute Data Type**
ub4 *

**Example**

```
OCIHandleAlloc(env,(void **) &pStatement, OCI_HTYPE_STMT, (size_t)0, (void **)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_CHNF_REGHANDLE

**Mode**
WRITE

**Description**
When this attribute is set to the appropriate subscription handle, execution of the query also creates the registration of the query for continuous query notification.

> **See Also:**
>
> Continuous Query Notification Attributes

**Attribute Data Type**
`OCISubscription *`

**Example**

```
/* Associate the statement with the subscription handle */
OCIAttrSet (stmthp, OCI_HTYPE_STMT, subscrhp, 0,
            OCI_ATTR_CHNF_REGHANDLE, errhp);
```

### OCI_ATTR_CQ_QUERYID

**Mode**
READ

**Description**
Obtains the query ID of a registered query after registration is made by the call to `OCIStmtExecute()`.

**Attribute Data Type**
`ub8 *`

### 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 Data Type**
`ub4 *`

### OCI_ATTR_ENV

**Mode**
READ

**Description**
Returns the environment context associated with the statement.

**Attribute Data Type**
`OCIEnv **`

### OCI_ATTR_FETCH_ROWID

**Mode**
READ/WRITE

**Description**
Specifies that the `ROWID`s are fetched after doing a define at `position` 0, and a `SELECT...FOR UPDATE` statement.

> **See Also:**
>
> Implicit Fetching of ROWIDs

**Attribute Data Type**
boolean */boolean

### OCI_ATTR_IMPLICIT_RESULT_COUNT

**Mode**
READ

**Description**
Returns the total number of implicit results available on the top-level OCI statement handle.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_NUM_DML_ERRORS

**Mode**
READ

**Description**
Returns the number of errors in the DML operation.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_PARAM_COUNT

**Mode**
READ

**Description**
Gets the number of columns in the select-list for the statement associated with the statement handle.

**Attribute Data Type**
`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((void *)stmthp, OCI_HTYPE_STMT, (void *)&parmcnt,
                      (ub4 *)0, OCI_ATTR_PARAM_COUNT, errhp));
```

```
/* Go through the column list and retrieve the data type of each column. You
   start from pos = 1 */
for (i = 1; i <= parmcnt; i++)
{
  /* Get parameter for column i */
  checkerr(errhp, OCIParamGet((void *)stmthp, OCI_HTYPE_STMT, errhp,
           (void **)&colhd, i));

  /* Get data-type of column i */
  type = 0;
  checkerr(errhp, OCIAttrGet((void *)colhd, OCI_DTYPE_PARAM,
          (void *)&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 Data Type
`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 the memory level 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 Data Type
`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 Data Type
`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 nonscrollable 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. Because they are forward sequential only, this also represents the highest row number seen by the application.

For scrollable cursors, `OCI_ATTR_ROW_COUNT` represents the maximum (absolute) row number fetched into the user buffers. Because 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.

Beginning with Oracle Database Release 12.1, using the attribute `OCI_ATTR_UB8_ROW_COUNT` is preferred to using the attribute `OCI_ATTR_ROW_COUNT` if row count values can exceed the value of `UB4MAXVAL` for an OCI application. If the row count exceeds the value of `UB4MAXVAL` and the application uses the attribute `OCI_ATTR_ROW_COUNT`, a call using `OCIAttrGet()` will return an error.

### Attribute Data Type

`ub4 *`

## OCI_ATTR_DML_ROW_COUNT_ARRAY

### Mode

READ

### Description

Returns an array of row counts affected by each iteration of an array DML statement. The row count for iteration *i* can be obtained by looking up array[*i*-1].

Without `OCI_BATCH_ERRORS` mode, OCIStmtExecute() stops execution with the first erroneous iteration. In such a scenario, the array returned by the `OCI_ATTR_DML_ROW_COUNT_ARRAY` attribute only contains valid row counts up to and including the last successful iteration. When `OCI_RETURN_ROW_COUNT_ARRAY` mode is used in conjunction with `OCI_BATCH_ERRORS` mode, the returned row-count array contains the actual number of rows affected per successful iteration and 0 for iterations that resulted in errors.

This attribute works only when the statement is executed in mode `OCI_RETURN_ROW_COUNT_ARRAY` at the time of using OCIStmtExecute().

Use this attribute only after an array DML operation and while using `OCI_RETURN_ROW_COUNT_ARRAY` mode in OCIStmtExecute().

Any attempt to query this attribute after any other operation (other than an array DML) or without passing this mode will result in an `OCI_ERROR` (`ORA-24349`).

### Attribute Data Type

`ub8 *`

### Example

```
int deptarray[]={10,20,30};
int iters = 3;
ub8 *rowcounts;
ub4 rowCountArraySize;
/*Statement prepare */
text *updatesal = (text *)"UPDATE EMP set sal = sal+100 where deptno = :dept"
OCIStmtPrepare2 ((OCISvcCtx *)svchp,(OCIStmt **)&stmthp,
(OCIError *)errhp, (text *)updatesal, (ub4)sizeof(updatesal)-1,
(oratext *)NULL, (ub4) 0, (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
/*Array bind*/
OCIBindByPos (stmthp, &bndhp, errhp, 1, deptarray,  sizeof(deptarray[0]),
         SQLT_INT, (ub2 *) 0, (ub2 *) 0, (ub4)   0, (ub4)   0, (ub4 *) 0,
         (ub4)   OCI_DEFAULT);
/* Pass new MODE for Array DML rowcounts; also, if an error occurred for any
iteration and you want to get the number of rows updated for the rest of the
iterations.*/
OCIStmtExecute(svchp, stmthp, errhp, iters,  (ub4) 0, 0, 0, OCI_BATCH_ERRORS |
OCI_RETURN_ROWCOUNT_ARRAY);
OCIAttrGet (stmthp, (ub4) OCI_HTYPE_STMT,
            (ub8 *)&rowcounts, &rowCountArraySize,
            OCI_ATTR_DML_ROW_COUNT_ARRAY, errhp);
```

### OCI_ATTR_ROWID

**Mode**
READ

**Description**
Returns the `ROWID` descriptor allocated with `OCIDescriptorAlloc()`.

> **See Also:**
>
> Positioned Updates and Deletes and ROWID Descriptor

**Attribute Data Type**
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 nonscrollable statement handles.

**Attribute Data Type**
`ub4 *`

**Example**

```
ub4 rows;
ub4 sizep = sizeof(ub4);
OCIAttrGet((void *) stmhp, (ub4) OCI_HTYPE_STMT,
           (void *)& rows, (ub4 *) &sizep, (ub4)OCI_ATTR_ROWS_FETCHED,
           errhp);
```

### OCI_ATTR_SQL_ID

**Mode**
READ

**Dexcription**
Gets the `SQL_ID` for a specific SQL statement from the server and makes it available on the client in a statement handle.

**Attribute Data Type**
`ub4 *`

**Example**
In an `OCIStmtPrepare2()`, specify `OCI_PREP2_GET_SQL_ID` as the mode. This `OCI_PREP2_GET_SQL_ID` mode should be ORed with any other mode. For example:

```
OCIStmtPrepare2((dvoid *)ctxptr->svchp, &stmthp,
                                (dvoid *)ctxptr->errhp,
                                insStmt, sizeof(insStmt),
                                (const oratext *)0, (ub4)0,
                                OCI_NTV_SYNTAX,
                                OCI_DEFAULT | OCI_PREP2_GET_SQL_ID);
```

You can do a bind or define operation and then execute on the statement.
The following code example shows how to obtain the `SQL_ID`.

```
  ub4         sqlidLen;
  oratext    *sqlid;
                OCIAttrGet(stmthp, OCI_HTYPE_STMT,
                                &sqlid, (ub4 *)&sqlidLen,
                                OCI_ATTR_SQL_ID,
                                ctxptr->errhp);
```

### OCI_ATTR_SQLFNCODE

**Mode**
READ

**Description**
Returns the function code of the SQL command associated with the statement.

**Attribute Data Type**
`ub2 *`

**Notes**
Table A-1 lists the SQL command codes.

**Table A-1 Function Code of the SQL Command Associated with the SQL Statement**

| Function Code | SQL Statement | Function Code | SQL Statement | Function Code | SQL Statement |
|---|---|---|---|---|---|
| 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 |

**Table A-1 (Cont.) Function Code of the SQL Command Associated with the SQL Statement**

| Function Code | SQL Statement | Function Code | SQL Statement | Function Code | SQL Statement |
|---|---|---|---|---|---|
| 05 | UPDATE | 47 | ALTER ROLLBACK SEGMENT | 89 | DROP BITMAPFILE |
| 06 | DROP ROLE | 48 | DROP ROLLBACK SEGMENT | 90 | SET CONSTRAINTS |
| 07 | DROP VIEW | 49 | CREATE TABLESPACE | 91 | CREATE FUNCTION |
| 08 | DROP TABLE | 50 | ALTER TABLESPACE | 92 | ALTER FUNCTION |
| 09 | DELETE | 51 | DROP TABLESPACE | 93 | DROP FUNCTION |
| 10 | CREATE VIEW | 52 | ALTER SESSION | 94 | CREATE PACKAGE |
| 11 | DROP USER | 53 | ALTER USER | 95 | ALTER PACKAGE |
| 12 | CREATE ROLE | 54 | COMMIT (WORK) | 96 | DROP PACKAGE |
| 13 | CREATE SEQUENCE | 55 | ROLLBACK | 97 | CREATE PACKAGE BODY |
| 14 | ALTER SEQUENCE | 56 | SAVEPOINT | 98 | ALTER PACKAGE BODY |
| 15 | (NOT USED) | 57 | CREATE CONTROL FILE | 99 | DROP PACKAGE BODY |
| 16 | DROP SEQUENCE | 58 | ALTER TRACING | 157 | CREATE DIRECTORY |
| 17 | CREATE SCHEMA | 59 | CREATE TRIGGER | 158 | DROP DIRECTORY |
| 18 | CREATE CLUSTER | 60 | ALTER TRIGGER | 159 | CREATE LIBRARY |
| 19 | CREATE USER | 61 | DROP TRIGGER | 160 | CREATE JAVA |
| 20 | CREATE INDEX | 62 | ANALYZE TABLE | 161 | ALTER JAVA |
| 21 | DROP INDEX | 63 | ANALYZE INDEX | 162 | DROP JAVA |
| 22 | DROP CLUSTER | 64 | ANALYZE CLUSTER | 163 | CREATE OPERATOR |
| 23 | VALIDATE INDEX | 65 | CREATE PROFILE | 164 | CREATE INDEXTYPE |
| 24 | CREATE PROCEDURE | 66 | DROP PROFILE | 165 | DROP INDEXTYPE |
| 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 |

**Table A-1 (Cont.) Function Code of the SQL Command Associated with the SQL Statement**

| Function Code | SQL Statement | Function Code | SQL Statement | Function Code | SQL Statement |
|---|---|---|---|---|---|
| 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 Data Type**
`oratext *`

### OCI_ATTR_STMTCACHE_CBKCTX

**Mode**
READ/WRITE

**Description**
Used to get and set the application's opaque context on the statement handle. This context can be of any type that the application defines. It is primarily used for encompassing the bind and define buffer addresses.

**Attribute Data Type**
`void *`

### OCI_ATTR_STMT_IS_RETURNING

**Mode**
READ

**Description**
Determines whether the given SQL Statement has a `RETURNING INTO` clause.

**Attribute Data Type**
`ub1`

**Example**

```
returning_into_clause = OCIAttrGet ( stmthp, OCI_HTYPE_STMT,
(ub1*)&isReturning, NULL, OCI_ATTR_STMT_IS_RETURNING, errhp);
```

### 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 you are in the middle of a fetch-execute cycle, then you 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 Data Type**
`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)
- `OCI_STMT_CALL` (PL/SQL statement)
- `OCI_STMT_MERGE` (PL/SQL statement)

**Attribute Data Type**
`ub2 *`

### OCI_ATTR_UB8_ROW_COUNT (Recommended over OCI_ATTR_ROW_COUNT)

**Mode**
READ

**Description**
For `SELECT` statements, returns the cumulative number of rows fetched from a result set. For `INSERT`, `UPDATE`, and `DELETE` statements, this attribute returns the number of rows processed by the statement. The default value is 1.
For non-scrollable cursors, `OCI_ATTR_UB8_ROW_COUNT` is the total number of rows fetched into user buffers with the `OCIStmtFrtch()` or OCIStmtFetch2() calls issued since this statement handle was executed. For these non-scrollable cursors, this also represents the highest row number seen by the application.
If using the attribute `OCI_ATTR_ROW_COUNT` and the row count returned is larger than `UB4MAXVAL`, then one or both of the following errors may be returned:

```
ORA-03148. OCI_ATTR_ROW_COUNT cannot see row counts larger than UB4MAXVAL
```

**Attribute Data Type**
`ub8 *`

## A.10 Bind Handle Attributes

Lists and describes bind handle attributes.

The following attributes are used for the bind handle.

**OCI_ATTR_CHAR_COUNT**

**Mode**
WRITE

**Description**
Sets the number of characters in character type data.

> **See Also:**
>
> Buffer Expansion During OCI Binding

**Attribute Data Type**
`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 causes 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 Data Type**
`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, the user must set the character set ID to `OCI_UTF16ID` (replaces the deprecated `OCI_UC2SID`, which is retained for backward compatibility). The bind value buffer is assumed to be a `utext` buffer, so 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 must 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` before setting `OCI_ATTR_CHARSET_FORM` causes unexpected results.

> **See Also:**
>
> Character Conversion in OCI Binding and Defining

**Attribute Data Type**
`ub2 *`

### OCI_ATTR_COLLATION_ID

**Mode**
READ/WRITE

**Description**
Sets the derived collation of a placeholder expression corresponding to this bind handle in a SQL statement. The attribute is relevant only for bind variables with character data types.
You can set the `OCI_ATTR_COLLATION_ID` attribute on a bind variable handle to any of the supported collation IDs. Collation IDs of both named collations and pseudo-collations are allowed. The attribute value is passed to the server with other bind information and the corresponding bind variable reference, formally known as SQL placeholder expression, assumes the provided collation at the coercibility level 0. If the attribute value is `OCI_COLLATION_NONE` (the default value), the collation of the bind variable is `USING_NLS_COMP` at the coercibility level 4.
OCI does not check whether the collation is valid for a given data type of a bind variable. If `OCI_ATTR_COLLATION_ID` is set for a non-character data type variable, it is ignored by the server.
Collation of bind variables is currently ignored in PL/SQL expressions. For forward compatibility reasons, the `OCI_ATTR_COLLATION_ID` attribute should not be set for bind variables passed to an anonymous PL/SQL block, unless the variables are referenced exclusively in SQL statements.

> **Note:**
>
> You can use the SQL built-in functions `NLS_COLLATION_ID` and `NLS_COLLATION_NAME` to map between values of the attribute, which are collation IDs, and collation names used in SQL syntax.

> **See Also:**
>
> - Column Attributes
> - *Oracle Database Globalization Support Guide* for information on how this set collation influences query processing

**Attribute Data Type**
`ub4 *`

### 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:**
>
> About Using the OCI_ATTR_MAXCHAR_SIZE Attribute

**Attribute Data Type**
`sb4 *`

### OCI_ATTR_MAXDATA_SIZE

**Mode**
READ/WRITE

**Description**
Sets the maximum number of bytes allowed in the buffer on the server side to accommodate client-side bind data after character set conversions.

> **See Also:**
>
> About Using the OCI_ATTR_MAXDATA_SIZE Attribute

**Attribute Data Type**
`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 must be changed, first perform a rebind/redefine operation, and then reset the two attributes in order.

**Attribute Data Type**
`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 must be changed, first perform a rebind/redefine operation, and then reset the two attributes in order.

**Attribute Data Type**
`sb2 *`

### OCI_ATTR_ROWS_RETURNED

**Mode**
READ

**Description**
This attribute returns the number of rows that will be returned in the current iteration when you are in the `OUT` callback function for binding a DML statement with a `RETURNING` clause.

**Attribute Data Type**
`ub4 *`

## A.11 Define Handle Attributes

Lists and describes define handle attributes.

The following attributes are used for the define handle.

### OCI_ATTR_CHAR_COUNT

**Mode**
WRITE

**Description**
This attribute is deprecated.
Sets the number of characters in 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 Data Type**
`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 causes 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 Data Type**
`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 must set the character set IDOTT to `OCI_UTF16ID`. The define value buffer is assumed to be a `utext` buffer, so 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 must 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` before setting `OCI_ATTR_CHARSET_FORM` causes unexpected results.

> **See Also:**
>
> Character Conversion in OCI Binding and Defining

**Attribute Data Type**
`ub2 *`

### OCI_ATTR_LOBPREFETCH_LENGTH

**Mode**
READ/WRITE

**Description**
Specifies if the LOB length and the chunk size for the LOB locators should be prefetched along with the descriptor if the column is a LOB column. Setting it to `TRUE` will enable the prefetching and this attribute setting is also required for prefetching the LOB data using the attribute `OCI_ATTR_LOBPREFETCH_SIZE`.

**Attribute Data Type**
`boolean */boolean`

### OCI_ATTR_LOBPREFETCH_SIZE

**Mode**
READ/WRITE

**Description**
Overrides the default cache buffer size for the LOB locators to be fetched from a particular column along with the descriptor. This requires the attribute `OCI_ATTR_LOBPREFETCH_LENGTH` to be set to `TRUE`.

**Attribute Data Type**
`ub4 */ub4`

### OCI_ATTR_MAXCHAR_SIZE

**Mode**
WRITE

**Description**
Specifies the maximum number of characters that the client application allows in the define buffer.

> **See Also:**
>
> About Using the OCI_ATTR_MAXCHAR_SIZE Attribute

**Attribute Data Type**
`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 must be changed, first perform a rebind/redefine operation, and then reset the two attributes in order.

**Attribute Data Type**
`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 must be changed, first perform a rebind/redefine operation, and then reset the two attributes in order.

**Attribute Data Type**
`sb2 *`

## A.12 Describe Handle Attributes

Lists and describes describe handle attributes.

The following attributes are used for the describe handle.

**OCI_ATTR_PARAM**

**Mode**
READ

**Description**
Points to the root of the description. Used for subsequent calls to OCIAttrGet() and OCIParamGet().

**Attribute Data Type**
`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 Data Type**
`ub4 *`

### OCI_ATTR_SHOW_INVISIBLE_COLUMNS

**Mode**
WRITE

**Description**
This attribute requests OCIDescribeAny() to also get the metadata for invisible columns. You can use OCIAttrGet() to determine whether a column is invisible or not.

**Attribute Data Type**
`boolean *`

**Example**

```
boolean showInvisibleCols = TRUE;
ub1 colInvisible[MAX_COLS];
OCIAttrSet(descHandle, OCI_HTYPE_DESCRIBE, &showInvisibleCols, 0,
            OCI_ATTR_SHOW_INVISIBLE_COLUMNS, errHandle);
if (rc = OCIDescribeAny(svcHandle, errHandle, (dvoid*)table,
                        strlen(table), OCI_OTYPE_NAME, 1,
                        OCI_PTYPE_TABLE, descHandle))
{
  OCIHandleFree(descHandle, OCI_HTYPE_DESCRIBE);
  return OCI_ERROR;
}

/* Get the number of columns. */

OCIAttrGet(parHandle, OCI_DTYPE_PARAM, &nCols, 0,
           OCI_ATTR_NUM_COLS, errHandle);

/* Get the column list. */

OCIAttrGet(parHandle, OCI_DTYPE_PARAM, &lstHandle, 0,
           OCI_ATTR_LIST_COLUMNS, errHandle);

/* Loop through the columns. */
for (i = 1; i <= nCols; i++)
{
  OCIParamGet(lstHandle, OCI_DTYPE_PARAM, errHandle,
              (dvoid*)&colHandle, i);
  OCIAttrGet(colHandle, OCI_DTYPE_PARAM, &colName[i-1], &len,
              OCI_ATTR_NAME, errHandle);
  OCIAttrGet(colHandle, OCI_DTYPE_PARAM, &(colType[i-1]), 0,
              OCI_ATTR_DATA_TYPE, errHandle);
  OCIAttrGet(colHandle, OCI_DTYPE_PARAM, &colInvisible[i-1], 0,
              OCI_ATTR_SHOW_INVISIBLE_COLUMNS, errHandle);
  if (colInvisible & OCI_ATTR_SHOW_INVISIBLE_COLUMNS)
     printf("Column is invisible\n");
}
```

# A.13 Parameter Descriptor Attributes

Lists and describes parameter descriptor attributes.

The following attributes are used for the parameter descriptor.

> **See Also:**
>
> Describing Schema Metadata for a detailed list of parameter descriptor attributes.

## A.14 Shard Instance Descriptor Attributes

Lists and describes shard instance descriptor attributes.

This descriptor is returned by `OCIShardInstancesGet()` used only by custom pools, which essentially contains the instance name where the data matching the provided sharding key resides, and an indication if that shard is writable on that instance.

The following attributes are used for the shard instance descriptor.

**OCI_ATTR_INSTNAME**

**Mode**
READ

**Description**
This attribute can be used to find the shard instance name out of a shard instance descriptor that is filed by an `OCIShardInstancesGet()` call. This attribute can also be used on the service context to return the shard instance name for a given connection. When called as an event handle attribute, `OCIAttrGet()` retrieves the name of the instance that has been affected by this event. This is also a server handle attribute.

**Attribute Data Type**
`oratext **`

**Example**

```
OCIAttrGet(shardInstance,
          (ub4) OCI_DTYPE_SHARD_INST,(dvoid *)&iName, (ub4 *)&iNameLen,
          OCI_ATTR_INSTNAME, (OCIError *)errhp);
```

**OCI_ATTR_SHARD_HAS_WRITABLECHUNK**

**Mode**
READ

**Description**
Returns `TRUE` if the shard instance has writable chunks. For custom pool implementations, this helps the pool to determine if a connection to that particular shard instance can be dispensed. If the application can work with read only shards instead of writable chunks, the pool can dispense it; otherwise, it can implement a retrial logic to try `OCIShardInstancesGet()` until it returns shard instance information having writable chunks. This feature is helpful during chunk migrations.

**Attribute Data Type**
`boolean *`

## A.15 SODA Document Handle Attributes

The `OCISodaDoc` document handle has the following attributes. The key, last-modified time stamp, creation time stamp, version, content, and media type attributes represent document components. JSON charset ID is a read-only character set ID set by SODA. Detect JSON Encoding is a flag for working with Unicode encoding JSON content.

The following attributes are document components:

- Key — `OCI_ATTR_SODA_KEY`
- Last-modified time stamp — `OCI_ATTR_SODA_LASTMOD_TIMESTAMP`
- Creation time stamp — `OCI_ATTR_SODA_CREATE_TIMESTAMP`
- Version — `OCI_ATTR_SODA_VERSION`
- Content — `OCI_ATTR_SODA_CONTENT`
- Media type ("application/json" for JSON documents) — `OCI_ATTR_SODA_MEDIA_TYPE`

The following attributes are not document components, but instead used to describe other characteristics of the document:

- JSON character set ID — `OCI_ATTR_SODA_JSON_CHARSET_ID`
- Detect JSON encoding — `OCI_ATTR_SODA_DETECT_JSON_ENC`

### OCI_ATTR_SODA_KEY

**Mode**
READ/WRITE

**Description**
The unique key for the document.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_LASTMOD_TIMESTAMP

**Mode**
READ

**Description**
The last-modified time stamp for the document in ISO format. The value is `NULL` if the metadata of the collection does not support it.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_CREATE_TIMESTAMP

**Mode**
READ

**Description**
The creation time stamp for the document in ISO format. The value is `NULL` if the metadata of the collection does not support it.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_VERSION

**Mode**
READ

**Description**
The document version. The value is `NULL` if the metadata of the collection does not support it.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_CONTENT

**Mode**
READ/WRITE

**Description**
The document content. Note that for `OCISodaInsertAndGet()`, `OCISodaInsertAndGetWithCtnt()`, and `OCISodaReplOneAndGetWithKey()`, these functions return the handle to the result document containing all components except the content, so this attribute is not needed for these calls.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_JSON_CHARSET_ID

**Mode**
READ

**Description**
The character set ID of the document's content. This could potentially be different from the local (client-side) character set ID.

**Attribute Data Type**
`ub2`

### OCI_ATTR_SODA_DETECT_JSON_ENC

**Mode**
READ/WRITE

**Description**
Returns `TRUE` if the document is in detect encoding mode.

**Attribute Data Type**
`boolean`

**OCI_ATTR_SODA_MEDIA_TYPE**

**Mode**
READ/WRITE

**Description**
The media type for the document.

**Attribute Data Type**
`oratext *`

## A.16 SODA Collection Handle Attributes

Lists and describes the OCI SODA collection handle attributes.

Collection metadata is composed of multiple components. `OCI_ATTR_SODA_DESCRIPTOR` is a special case. When you pass `OCI_ATTR_SODA_DESCRIPTOR` to `OCIAttrGet()`, you get metadata for the collection in one chunk, in JSON form. In addition, `OCI_ATTR_SODA_COLL_NAME`, the collection name, is not part of collection metadata (it is not a collection metadata component). It is stored separately in SODA, so it is also a special case. If you use `OCI_ATTR_SODA_DESCRIPTOR` to fetch collection metadata in JSON, the collection name is not included.

The rest of these attributes fetch individual metadata components. They are provided for convenience (alternatively, you could get collection metadata using `OCI_ATTR_SODA_DESCRIPTOR`. That metadata contains all of the components, but it is in JSON form, so your application might need to parse the JSON to retrieve values of specific components). All of these attributes are `READ` only and allow you to examine the metadata for your existing collection.

> **See Also:**
>
> *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for reference information about these SODA collection metadata components. All the metadata components that correspond to these collection handle attributes (except for `OCI_ATTR_SODA_DESCRIPTOR` and `OCI_ATTR_SODA_COLL_NAME`), are described in this chapter.

The following is a list of all collection attributes followed by descriptions of each.

- Collection Name — `OCI_ATTR_SODA_COLL_NAME`
- Descriptor — `OCI_ATTR_SODA_DESCRIPTOR`
- Schema — `OCI_ATTR_SODA_SCHEMA`
- Table Name — `OCI_ATTR_SODA_TABLE_NAME`
- View Name — `OCI_ATTR_SODA_VIEW_NAME`
- Key Column Name — `OCI_ATTR_SODA_KEY_COL_NAME`
- Key Column SQL Type — `OCI_ATTR_SODA_KEY_SQL_TYPE`
- Key Column Max Length — `OCI_ATTR_SODA_KEY_MAX_LEN`

- Key Column Assignment Method — `OCI_ATTR_SODA_KEY_ASSIGN_METHOD`
- Key Column Sequence Name — `OCI_ATTR_SODA_KEY_SEQ_NAME`
- Content Column Name — `OCI_ATTR_SODA_CTNT_COL_NAME`
- Content Column SQL Type — `OCI_ATTR_SODA_CTNT_SQL_TYPE`
- Content Column Max Length — `OCI_ATTR_SODA_CTNT_MAX_LEN`
- Content Column JSON Validation — `OCI_ATTR_SODA_CTNT_VALIDATION`
- Content Column SecureFiles LOB Compression — `OCI_ATTR_SODA_CTNT_COMPRESS`
- Content Column SecureFiles LOB Cache — `OCI_ATTR_SODA_CTNT_CACHE`
- Content Column SecureFiles LOB Encryption — `OCI_ATTR_SODA_CTNT_ENCRYPT`
- Version Column Name — `OCI_ATTR_SODA_VERSION_COL_NAME`
- Version Generation Method — `OCI_ATTR_SODA_VERSION_METHOD`
- Last-Modified Time Stamp Column Name — `OCI_ATTR_SODA_MODTIME_COL_NAME`
- Last-Modified Column Index Name — `OCI_ATTR_SODA_MODTIME_INDEX`
- Creation Time Stamp Column Name — `OCI_ATTR_SODA_CRTIME_COL_NAME`
- Media Type Column Name — `OCI_ATTR_SODA_MTYPE_COL_NAME`
- Read Only — `OCI_ATTR_SODA_READONLY`

### OCI_ATTR_SODA_COLL_NAME

**Mode**
READ

**Description**
Specifies the name of the collection.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_DESCRIPTOR

**Mode**
READ

**Description**
Specifies the descriptor to which the collection is mapped.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_SCHEMA

**Mode**
READ

**Description**
Specifies the name of the Oracle Database schema that owns the table or view to which the collection is mapped.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_TABLE_NAME

**Mode**
READ

**Description**
Specifies the name of the table to which the collection is mapped.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_VIEW_NAME

**Mode**
READ

**Description**
Specifies the name of the view to which the collection is mapped.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_KEY_COL_NAME

**Mode**
READ

**Description**
Specifies the name of the column that stores the document key.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_KEY_SQL_TYPE

**Mode**
READ

**Description**
Specifies the SQL data type of the column that stores the document key. Valid values include:

- `SQLT_CHR` for `VARCHAR2`
- `SLQT_BIN` for `RAW`
- `SQLT_NUM` for `NUMBER`

**Attribute Data Type**
`ub1`

### OCI_ATTR_SODA_KEY_MAX_LEN

**Mode**
READ

**Description**
Specifies the maximum length of the key column in bytes. This component applies only to keys of type `VARCHAR2`. At least 32 bytes if key assignment method is `UUID` or `GUID`. See OCI_ATTR_SODA_KEY_ASSIGN_METHOD.

> **See Also:**
>
> OCI_ATTR_SODA_KEY_SQL_TYPE

**Attribute Data Type**
`ub4`

### OCI_ATTR_SODA_KEY_ASSIGN_METHOD

**Mode**
READ

**Description**
Specifies the OCI value for the corresponding value for this component method used to assign keys to objects that are inserted into the collection. `UUID`, `GUID`, `SEQUENCE`, and `CLIENT` are the corresponding valid values for this component, as it would appear in the JSON metadata. The valid values for this attribute include:

- `OCI_SODA_KEY_METHOD_UUID` for `UUID`, which is the corresponding value for this component, as it would appear in the JSON metadata.
- `OCI_SODA_KEY_METHOD_GUID` for `GUID`, which is the corresponding value for this component, as it would appear in the JSON metadata.
- `OCI_SODA_KEY_METHOD_SEQUENCE` for `SEQUENCE`, which is the corresponding value for this component, as it would appear in the JSON metadata.
- `OCI_SODA_KEY_METHOD_CLIENT` for `CLIENT`, which is the corresponding value for this component, as it would appear in the JSON metadata.

See *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for more information about key assignment methods for the collection metadata component Key Column Assignment Method.

**Attribute Data Type**
`ub1`

### OCI_ATTR_SODA_KEY_SEQ_NAME

**Mode**
READ

**Description**
Specifies the name of the database sequence that generates keys for documents that are inserted into a collection if the key assignment method is SEQUENCE.

**Note:**

If you drop a collection using SODA for OCI, the sequence used for key generation is *not* dropped. This is because it might not have been created using SODA for OCI. To drop the sequence, use SQL command `DROP SEQUENCE`, after first dropping the collection.

**See Also:**

- OCI_ATTR_SODA_KEY_ASSIGN_METHOD
- *Oracle Database SQL Language Reference* for information about `DROP SEQUENCE`
- *Oracle Database Concepts*

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_CTNT_COL_NAME

**Mode**
READ

**Description**
Specifies the name of the column that stores the document content.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_CTNT_SQL_TYPE

**Mode**
READ

**Description**
Specifies the SQL data type of the column that stores the document content. Valid values include:

- `SQLT_CHR` for `VARCHAR2`
- `SQLT_CLOB` for `CLOB`,
- `SQLT_BLOB` for `BLOB`

**Attribute Data Type**
`ub1`

### OCI_ATTR_SODA_CTNT_MAX_LEN

**Mode**
READ

**Description**
Specifies the maximum length of the content column in bytes. This component applies only to content of type `VARCHAR2`. The default value is 4000 bytes.

> **See Also:**
>
> - OCI_ATTR_SODA_CTNT_SQL_TYPE
> - *Oracle Database SQL Language Reference* for information about extended data types

**Attribute Data Type**
`ub4`

**OCI_ATTR_SODA_CTNT_VALIDATION**

**Mode**
READ

**Description**
Specifies the OCI value for the corresponding syntax to which JSON content must conform. The valid values for this attribute include:

- `OCI_SODA_JSON_VALIDATION_STRICT` for `STRICT`, which is the corresponding value as it appears in the JSON collection metadata.

  See *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for more information about the `STRICT` syntax.
- `OCI_COLL_JSON_VALIDATION_LAX` for `LAX`, which is the corresponding value as it appears in the JSON collection metadata.

  See *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for more information about the `LAX` syntax.
- `OCI_SODA_JSON_VALIDATION_STD` for `STANDARD`, which is the corresponding value as it appears in the JSON collection metadata.

  See *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for more information about the `STANDARD` syntax.

**Attribute Data Type**
`ub4`

**OCI_ATTR_SODA_CTNT_COMPRESS**

**Mode**
READ

**Description**
Specifies the OCI value for the corresponding value for the SecureFiles LOB compression setting as it appears in the JSON collection metadata. The valid values for this attribute include:

- `OCI_SODA_LOB_COMPRESS_NONE` for `NONE`, which is the corresponding value as it appears in the JSON collection metadata.
- `OCI_SODA_LOB_COMPRESS_HIGH` for `HIGH`, which is the corresponding value as it appears in the JSON collection metadata.
- `OCI_SODA_LOB_COMPRESS_MEDIUM` for `MEDIUM`, which is the corresponding value as it appears in the JSON collection metadata.
- `OCI_SODA_LOB_COMPRESS_LOW` for `LOW`, which is the corresponding value as it appears in the JSON collection metadata.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about SecureFiles LOB storage

**Attribute Data Type**
`ub1`

### OCI_ATTR_SODA_CTNT_CACHE

**Mode**
READ

**Description**
Specifies the SecureFiles LOB cache setting.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about SecureFiles LOB storage

**Attribute Data Type**
`boolean`

### OCI_ATTR_SODA_CTNT_ENCRYPT

**Mode**
READ

**Description**
Specifies the OCI value for the corresponding value for the SecureFiles LOB encryption setting as it appears in the JSON collection metadata. The valid values for this attribute include:

- `OCI_SODA_LOB_ENCRYPT_NONE` for `NONE`, which is the corresponding value as it appears in the JSON collection metadata.
- `OCI_SODA_LOB_ENCRYPT_3DES168` for `3DES168`, which is the corresponding value as it appears in the JSON collection metadata.

- `OCI_SODA_LOB_ENCRYPT_AES128` for `AES128`, which is the corresponding value as it appears in the JSON collection metadata.
- `OCI_SODA_LOB_ENCRYPT_AES192` for `AES192`, which is the corresponding value as it appears in the JSON collection metadata.
- `OCI_SODA_LOB_ENCRYPT_AES256` for `AES256`, which is the corresponding value as it appears in the JSON collection metadata.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about SecureFiles LOB storage

**Attribute Data Type**
`ub1`

### OCI_ATTR_SODA_VERSION_COL_NAME

**Mode**
READ

**Description**
Specifies the name of the column that stores the document version.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_VERSION_METHOD

**Mode**
READ

**Description**
Specifies the OCI value for the corresponding method used to compute version values for objects when they are inserted into a collection or replaced as it appears in the JSON collection metadata. The valid values for this attribute include:

- `OCI_SODA_VERSION_UUID` for `UUID`, which is the corresponding value as it appears in the JSON collection metadata
- `OCI_SODA_VERSION_TIMESTAMP` for `TIMESTAMP`, which is the corresponding value as it appears in the JSON collection metadata
- `OCI_SODA_VERSION_MD5` for `MD5`, which is the corresponding value as it appears in the JSON collection metadata
- `OCI_SODA_VERSION_SHA256` for `SHA256`, which is the corresponding value as it appears in the JSON collection metadata
- `OCI_SODA_VERSION_SEQUENTIAL` for `SEQUENTIAL`, which is the corresponding value as it appears in the JSON collection metadata
- `OCI_SODA_VERSION_NONE` for `NONE`, which is the corresponding value as it appears in the JSON collection metadata

See *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for more information about version generation methods for the collection metadata component Version Column Generation Method.

**Attribute Data Type**
`ub1`

### OCI_ATTR_SODA_MODTIME_COL_NAME

**Mode**
READ

**Description**
Specifies the name of the column that stores the last-modified time stamp of the document.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_MODTIME_INDEX

**Mode**
READ

**Description**
Specifies the name of the index on the last-modified column.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_CRTIME_COL_NAME

**Mode**
READ

**Description**
Specifies the name of the column that stores the creation time stamp of the document.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_SODA_MTYPE_COL_NAME

**Mode**
READ

**Description**
Specifies the name of the column that stores the media type of the document. A media type column is needed if the collection is to be heterogeneous, that is, it can store documents other than JSON.

**Attribute Data Type**
`oratext *`

**OCI_ATTR_SODA_READONLY**

**Mode**
READ

**Description**
Specifies whether the collection is read-only.

**Attribute Data Type**
`boolean`

## A.17 SODA Output Options Handle Attributes

Lists and describes the OCI SODA Output Options handle attributes. This handle is used to return the number of documents processed by a bulk operation. Currently, it is returned only by bulk insert methods.

**OCI_ATTR_SODA_DOC_COUNT**

**Mode**
READ

**Description**
Number of documents processed successfully by the statement.

**Attribute Data Type**
`ub8 *`

## A.18 SODA Operation Options Handle Attributes

Lists and describes the OCI SODA Operation Options handle attributes.

Operation options handle is used to drive read and write SODA operations. The attributes that can be set on operation options handle represent the conditions of the operation. After the attributes are set, the operation options handle is passed to SODA functions perform the actual read or write operations such as finds, replaces, removes, and so on.
The attributes listed in this section can be set on the `OCISodaOperationOptions` handle using the standard OCIAttrSet methods. However, if you want to set an array of keys on the `OCISodaOperationOptions` handle, then you must use `OCISodaOperKeysSet()` method.

**OCI_ATTR_SODA_KEY**

**Mode**
READ/WRITE

**Description**
Key to be used for the operation.

> **Note:**
>
> If you use function `OCIAttrSet()` to set attribute `OCI_ATTR_SODA_KEY` on an operation-options handle, and you also use function `OCISodaOperKeysSet()` to set multiple keys on the same handle, then only the latest of the two settings takes effect. The effect of the first function invoked is overridden by the effect of the second.

**Attribute Data Type**
`oratext **/oratext *`

### OCI_ATTR_SODA_VERSION

**Mode**
READ/WRITE

**Description**
Version to be used for the operation.

**Attribute Data Type**
`oratext **/oratext *`

### OCI_ATTR_SODA_FILTER

**Mode**
READ/WRITE

**Description**
Filter to be used for the operation.

**Attribute Data Type**
`oratext **/oratext *`

### OCI_ATTR_SODA_SKIP

**Mode**
READ/WRITE

**Description**
Number of document results to skip.

**Attribute Data Type**
`ub4 */ub4`

### OCI_ATTR_SODA_LIMIT

**Mode**
READ/WRITE

**Description**
Number of documents to limit in the operation results.

**Attribute Data Type**
`ub4 */ub4`

**Related Topics**

- 
- 
-

## A.19 LOB Descriptor and LOB Locator Attributes

Lists and describes LOB locator attributes.

The following attributes are used for the parameter descriptor.

**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, OCILobWrite2() or OCILobWrite() (deprecated) 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 the address of a `ub4` that has a value of 0; for example, declare the following:

```
ub4 lobEmpty = 0
```

Then they should pass the address: `&lobEmpty`.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_LOB_REMOTE**

**Mode**
READ

**Description**
Determines whether the LOB locator belongs to a local database table or a remote database table. The value `TRUE` indicates that the LOB locator is from a remote database table. The application must fetch the LOB descriptor from the database before using this attribute.
Only the server can set this value. If an application tries to set this attribute, then an `ORA-24315` error is raised.

**Attribute Data Type**
`boolean`

**OCI_ATTR_LOB_TYPE**

**Mode**
READ

**Description**
Gets the LOB type from the LOB descriptor.

**Attribute Data Type**
`ub2`

# A.20 Complex Object Attributes

Lists and describes complex object attributes.

The following attributes are used for complex objects.

> **See Also:**
>
> Complex Object Retrieval

## A.20.1 Complex Object Retrieval Handle Attributes

Lists and describes complex object retrieval handle attributes.

The following attributes are used for the complex object retrieval handle.

**OCI_ATTR_COMPLEXOBJECT_LEVEL**

**Mode**
WRITE

**Description**
The depth level for complex object retrieval.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE**

**Mode**
WRITE

**Description**
Whether to fetch collection attributes in an object type out-of-line.

**Attribute Data Type**
`ub1 *`

## A.20.2 Complex Object Retrieval Descriptor Attributes

Lists and describes complex object retrieval descriptor attributes.

The following attributes are used for the complex object retrieval descriptor.

**OCI_ATTR_COMPLEXOBJECTCOMP_TYPE**

**Mode**
WRITE

**Description**
A type of `REF` to follow for complex object retrieval.

**Attribute Data Type**
`void *`

**OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL**

**Mode**
WRITE

**Description**
Depth level for the following `REF`s of type `OCI_ATTR_COMPLEXOBJECTCOMP_TYPE`.

**Attribute Data Type**
`ub4 *`

# A.21 Database Advanced Queuing Descriptor Attributes

Lists and describes Database Advanced Queuing descriptor attributes

The following attributes are used for the database advanced queuing descriptor.

> **See Also:**
>
> *Oracle Database Advanced Queuing User's Guide*

## A.21.1 OCIAQEnqOptions Descriptor Attributes

Lists and describes 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 buffered message into a queue, by setting the `OCI_ATTR_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 Data Type**
`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 that 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 Data Type**
`OCIRaw *`

### OCI_ATTR_SEQUENCE_DEVIATION

**Mode**
READ/WRITE

**Description**
This feature is deprecated for new applications, but it is retained for compatibility.
It specifies whether the message being enqueued should be dequeued before other messages in the queue.

**Attribute Data Type**
`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_TRANSFORMATION

**Mode**
READ/WRITE

**Description**
The name of the transformation that must be applied before the message is enqueued into the database. The transformation must be created using `DBMS_TRANSFORM`.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_VISIBILITY

**Mode**
READ/WRITE

**Description**
Specifies the transactional behavior of the enqueue request.

**Attribute Data Type**
`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.

## A.21.2 OCIAQDeqOptions Descriptor Attributes

Lists and describes 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 Data Type**
`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 multiple messages satisfy the pattern, the order of dequeuing is undetermined.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_DEQ_MODE

**Mode**
READ/WRITE

**Description**
Specifies the locking behavior associated with the dequeue.

**Attribute Data Type**
`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 Data Type**
`OCIRaw *`

### OCI_ATTR_DEQCOND

**Mode**
READ/WRITE

**Description**
This attribute is a Boolean expression similar to the `WHERE` clause of a SQL query. This Boolean expression can include conditions on message properties, user data properties (object payloads only), and PL/SQL or SQL functions.
To specify dequeue conditions on a message payload (object payload), use attributes of the object type in clauses. You must prefix each attribute with `tab.user_data` as a qualifier to indicate the specific column of the queue table that stores the payload.
The attribute cannot exceed 4000 characters. If multiple messages satisfy the dequeue condition, then the order of dequeuing is indeterminate, and the sort order of the queue is not honored.

**Attribute Data Type**
`oratext *`

**Example**

```
checkerr(errhp, OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
         (dvoid *)"tab.priority between 2 and 4" ,
         strlen("tab.priority between 2 and 4"),
         OCI_ATTR_DEQCOND, errhp));
```

### 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_ATTR_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 Data Type**
`ub2`

### OCI_ATTR_NAVIGATION

**Mode**
READ/WRITE

**Description**
Specifies the position of the message that is retrieved. First, the position is determined. Second, the search criterion is applied. Finally, the message is retrieved.

**Attribute Data Type**
`ub4`

**Valid Values**
The only valid values are:

- `OCI_DEQ_FIRST_MSG` - Retrieves the first available message that matches the search criteria. This resets the position to the beginning of the queue.
- `OCI_DEQ_NEXT_MSG` - Retrieves the next available message that matches the search criteria. If the previous message belongs to a message group, AQ retrieves the next available message that 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_MULTI_GROUP` - Indicates that a call to `OCIAQDeqArray()` resets 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`, is defined. All messages belonging to the same transaction group have the same value for this message property.
- `OCI_DEQ_NEXT_MSG_MULTI_GROUP` - Indicates that a call to `OCIAQDeqArray()` dequeues 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`, is defined. All messages belonging to the same transaction group have the same value for this message property.

### OCI_ATTR_TRANSFORMATION

**Mode**
READ/WRITE

**Description**
The name of the transformation that must be applied after the message is dequeued but before returning it to the dequeuing application. The transformation must be created using `DBMS_TRANSFORM`.

**Attribute Data Type**
`oratext *`

### 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 Data Type**
`ub4`

**Valid Values**
The only valid values are:

- `OCI_DEQ_ON_COMMIT` - The dequeue is part of the current transaction. This is the default.
- `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 no message is currently available that matches the search criteria. This parameter is ignored if messages in the same group are being dequeued.

**Attribute Data Type**
`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.

## A.21.3 OCIAQMsgProperties Descriptor Attributes

Lists and describes 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 Data Type**
`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 Data Type**
`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 message ID (msgid) overrides the delay specification. A message enqueued with delay set is 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 the delay is set by the producer who enqueues the message.

**Attribute Data Type**
`sb4`

**Valid Values**
Any sb4 value is valid, but the following predefined constant is available:

> OCI_MSG_NO_DELAY - Indicates that the message is available for immediate dequeuing.

#### OCI_ATTR_ENQ_TIME

**Mode**
READ

**Description**
Specifies the time that the message was enqueued. This value is determined by the system and cannot be set by the user.

**Attribute Data Type**
`OCIDate`

#### OCI_ATTR_EXCEPTION_QUEUE

**Mode**
READ/WRITE

**Description**
Specifies the name of the queue to which the message is moved 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 is moved to the default exception queue associated with the queue table, and a warning is logged in the alert file. If the default exception queue is used, the parameter returns a `NULL` value at dequeue time.
This attribute must refer to a valid queue name.

**Attribute Data Type**
`oratext *`

#### OCI_ATTR_EXPIRATION

**Mode**
READ/WRITE

**Description**
Specifies the expiration of the message. It determines, in seconds, how long 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 is moved to the exception queue in the `EXPIRED` state.

**Attribute Data Type**
`sb4`

**Valid Values**
Any sb4 value is valid, but the following predefined constant is available:

OCI_MSG_NO_EXPIRATION - The message never expires.

### OCI_ATTR_MSG_DELIVERY_MODE

**Mode**
READ

**Description**
After a dequeue call, the OCI client can read the `OCI_ATTR_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 Data Type**
`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 Data Type**
`ub4`

**Valid Values**
Only the following values 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_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 Data Type**
`OCIRaw *`

### OCI_ATTR_PRIORITY

**Mode**
READ/WRITE

**Description**
Specifies the priority of the message. A smaller number indicates a higher priority. The priority can be any number, including negative numbers.
The default value is zero.

**Attribute Data Type**
`sb4`

### OCI_ATTR_RECIPIENT_LIST

**Mode**
WRITE

**Description**
This parameter is only valid for queues that allow multiple consumers. The default recipients are the queue subscribers. This parameter is not returned to a consumer at dequeue time.

**Attribute Data Type**
`OCIAQAgent **`

### OCI_ATTR_SENDER_ID

**Mode**
READ/WRITE

**Description**
Identifies the original sender of a message.

**Attribute Data Type**
`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 Data Type**
`oratext *`

## A.21.4 OCIAQAgent Descriptor Attributes

Lists and describes 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 Data Type**
`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 Data Type**
`oratext *`

**Valid Values**
Can be any Oracle Database 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 Data Type**
`ub1`

**Valid Values**
The only valid value is zero, which is also the default.

## A.21.5 OCIServerDNs Descriptor Attributes

Lists and describes 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 Data Type**
`ub2`

**OCI_ATTR_SERVER_DN**

**Mode**
READ/WRITE

**Description**
For read mode, this attribute returns the list of Oracle Database distinguished names that are already inserted into the descriptor.
For write mode, this attribute takes the distinguished name of an Oracle Database.

**Attribute Data Type**
`oratext **/oratext *`

# A.22 Subscription Handle Attributes

Lists and describes subscription handle attributes.

The following attributes are used for the subscription handle.

**See Also:**

- Publish-Subscribe Notification in OCI
- About Continuous Query Notification

**OCI_ATTR_SERVER_DNS**

**Mode**
READ/WRITE

**Description**
The distinguished names of the Oracle database that the client is interested in for the registration.

**Attribute Data Type**
`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 must be set before the subscription handle can be passed into the registration call `OCISubscriptionRegister()`.

**Attribute Data Type**
`ub4 (void *, OCISubscription *, void *, ub4, void *, ub4)`

### OCI_ATTR_SUBSCR_CQ_QOSFLAGS

**Mode**
WRITE

**Description**
Sets QOS (quality of service flags) specific to continuous query (CQ) notifications. For the possible values you can pass, see the section on using OCI subscription handle attributes for CQN in *Oracle Database Development Guide*.

**Attribute Data Type**
`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 must be set before the subscription handle can be passed into the registration call `OCI Subscription Register()`.

**Attribute Data Type**
`void *`

### OCI_ATTR_SUBSCR_HOSTADDR

**Mode**
READ/WRITE

**Description**
Before registering for notification using `OCISubscriptionRegister()`, specify the client IP (in either IPv4 or IPv6 format) of the listening endpoint of the OCI notification client to which the notification is sent. Enter either IPv4 addresses in dotted decimal format, for example, 192.0.2.34, or IPv6 addresses in hexadecimal format, for example, 2001:0db8:0000:0000:0217:f2ff:fe4b:4ced.

**See Also:**

*Oracle Database Net Services Administrator's Guide* for more information about the IPv6 format for IP addresses

**Attribute Data Type**
`text *`

**Example**

```
/* Set notification client address*/
text ipaddr[16] = "192.0.2.34";
(void) OCIAttrSet((dvoid *) envhp, (ub4) OCI_HTYPE_ENV,
                  (dvoid *) ipaddr, (ub4) strlen(ipaddr),
                  (ub4) OCI_ATTR_SUBSCR_HOSTADDR, errhp);
```

### OCI_ATTR_SUBSCR_IPADDR

**Mode**
READ/WRITE

**Description**
The client IP address (IPv4 or IPv6) on which an OCI client registered for notification listens, to receive notifications. For example, IPv4 address in dotted decimal format such as 192.1.2.34 or IPv6 address in hexadecimal format such as 2001:0db8:0000:0000:0217:f2ff:fe4b:4ced.

**See Also:**

*Oracle Database Net Services Administrator's Guide* for more information about the IPv6 format for IP addresses

**Attribute Data Type**
`oratext *`

### 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 must be specified as it is not assumed that the name is `NULL`-terminated. This is important because the name could contain multibyte characters. Clients can 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 Data Type**
`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 Data Type**
`ub4 *`

### OCI_ATTR_SUBSCR_NTFN_GROUPING_CLASS

**Mode**
READ/WRITE

**Description**
Notification grouping class. If set to 0 (the default) all other notification grouping attributes must be 0. It is implemented for time in the latest release and is the only current criterion for grouping. Can be set to `OCI_SUBSCR_NTFN_GROUPING_CLASS_TIME`.

> **Note:**
>
> `OCI_OBJECT` mode is required when using grouping notifications.

**Attribute Data Type**
`ub1 *`

### OCI_ATTR_SUBSCR_NTFN_GROUPING_REPEAT_COUNT

**Mode**
READ/WRITE

**Description**
How many times to do the grouping. Notification repeat count. Positive integer. Can be set to `OCI_NTFN_GROUPING_FOREVER` to send grouping notifications forever.

**Attribute Data Type**
`sb4 *`

### OCI_ATTR_SUBSCR_NTFN_GROUPING_START_TIME

**Mode**
READ/WRITE

**Description**
The time grouping starts. Set to a valid `TIMESTAMP WITH TIME ZONE`. The default is the current `TIMESTAMP WITH TIME ZONE`.

**Attribute Data Type**
`OCIDateTime */OCIDateTime **`

### OCI_ATTR_SUBSCR_NTFN_GROUPING_TYPE

**Mode**
READ/WRITE

**Description**
The format of the grouping notification: whether a summary of all events in the group or just the last event in the group. Use `OCIAttrSet()` to set to one of the following notification grouping types: `OCI_SUBSCR_NTFN_TYPE_SUMMARY` or `OCI_SUBSCR_NTFN_TYPE_LAST`. Summary of notifications is the default. The other choice is the last notification.

**Attribute Data Type**
`ub1 *`

### OCI_ATTR_SUBSCR_NTFN_GROUPING_VALUE

**Mode**
READ/WRITE

**Description**
Specifies the value for the grouping class. For time, this is the time-period of grouping notifications specified in seconds, that is, the time after which grouping notification is sent periodically until `OCI_ATTR_SUBSCR_NTFN_GROUPING_REPEAT_COUNT` is exhausted.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_SUBSCR_PAYLOAD

**Mode**
READ/WRITE

**Description**
Buffer that corresponds to the payload that must be sent along with the notification. The length of the buffer can also be specified in the same set attribute call. This attribute must be set before a post can be performed on a subscription. For the current release, only an untyped (`ub1 *`) payload is supported.

**Attribute Data Type**
`ub1 *`

### OCI_ATTR_SUBSCR_QOSFLAGS

**Mode**
READ/WRITE

**Description**
Quality of service levels of the server. The possible settings are:

- `OCI_SUBSCR_QOS_RELIABLE` - Reliable. If the database fails, it does not lose notification. Not supported for nonpersistent queues or buffered messaging.
- `OCI_SUBSCR_QOS_PURGE_ON_NTFN` - Once received, purge notification and remove subscription.
- `OCI_SUBSCR_QOS_PAYLOAD` - Payload notification.

**Attribute Data Type**
`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 email address (for example, xyz@oracle.com) to which the notification is sent. The validity of the email 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 is invoked when there is a notification. The subscriber must have appropriate permissions on the procedure for it to be executed.

> **See Also:**
>
> Notification Procedure for information about procedure definition

**Attribute Data Type**
`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`.
For event notification in XML presentation, set this attribute to `OCI_SUBSCR_PRES_XML`. XML presentation is limited to 2000 bytes. Otherwise, leave it unset or set it to `OCI_SUBSCR_PRES_DEFAULT`.

**Attribute Data Type**
`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 wants to receive the event notification, then you should set this attribute to `OCI_SUBSCR_PROTO_OCI`.
If you want an email 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 an 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 Data Type**
`ub4 *`

### OCI_ATTR_SUBSCR_TIMEOUT

**Mode**
READ/WRITE

**Description**
Registration timeout interval in seconds. If 0 or not specified, then the registration is active until the subscription is explicitly unregistered.

**Attribute Data Type**
`ub4 *`

## A.22.1 Continuous Query Notification Attributes

Lists and describes continuous query notification attributes.

The following attributes are used for continuous query notification.

**OCI_ATTR_CHNF_CHANGELAG**

**Mode**
WRITE

**Description**
The number of transactions that the client is to lag in continuous query notifications.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_CHNF_OPERATIONS**

**Mode**
WRITE

**Description**
Used to filter notifications based on operation type.

> **See Also:**
>
> About Continuous Query Notification for details about the flag values

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_CHNF_ROWIDS**

**Mode**
WRITE

**Description**
If `TRUE`, the continuous query notification message includes row-level details, such as operation type and `ROWID`. The default is `FALSE`.

**Attribute Data Type**
`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 Data Type**
`OCIColl **`

## A.22.2 Continuous Query Notification Descriptor Attributes

Lists and describes continuous query notification descriptor attributes.

The following attributes are used for the continuous query notification descriptor.

### OCI_ATTR_CHDES_DBNAME

**Mode**
READ

**Description**
Name of the database.

**Attribute Data Type**
`oratext **`

### OCI_ATTR_CHDES_NFTYPE

**Mode**
READ

**Description**
Flags describing the notification type.

> **See Also:**
>
> About Continuous Query Notification for the flag values

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_CHDES_ROW_OPFLAGS

**Mode**
READ

**Description**
Operation type: `INSERT`, `UPDATE`, `DELETE`, or `OTHER`.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_CHDES_ROW_ROWID

**Mode**
READ

**Description**
String representation of a `ROWID`.

**Attribute Data Type**
`oratext **`

### OCI_ATTR_CHDES_TABLE_CHANGES

**Mode**
READ

**Description**
A collection type describing operations on tables. Each element of the collection is a table continuous query descriptors (`OCITableChangeDesc *`) of type `OCI_DTYPE_TABLE_CHDES` that has the attributes that begin with `OCI_ATTR_CHDES_TABLE`. See the following entries.

**Attribute Data Type**
`OCIColl **`

### OCI_ATTR_CHDES_TABLE_NAME

**Mode**
READ

**Description**
Schema and table name. `HR.EMPLOYEES`, for example.

**Attribute Data Type**
`oratext **`

### OCI_ATTR_CHDES_TABLE_OPFLAGS

**Mode**
READ

**Description**
Flags describing the operations on the table.

**Attribute Data Type**
`ub4 *`

> **See Also:**
>
> *Oracle Database Development Guide* for information about the flag values for the `OCI_DTYPETABLE_CHDES` continuous query notification descriptor

### 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 continuous query descriptor (`OCIRowChangeDesc *`) of type `OCI_DTYPE_ROW_CHDES` that has the attributes `OCI_ATTR_CHDES_ROW_OPFLAGS` and `OCI_ATTR_CHDES_ROW_ROWID`.

**Attribute Data Type**
`OCIColl **`

### OCI_ATTR_CHDES_XID

**Mode**
READ

**Description**
The transaction ID of the message.

**Attribute Data Type**
`OCIRaw *`

> **See Also:**
>
> *Oracle Database Development Guide* for more information.

## A.22.3 Notification Descriptor Attributes

Lists and describes notification descriptor attributes.

The following are attributes of the descriptor `OCI_DTYPE_AQNFY`.

**OCI_ATTR_AQ_NTFN_GROUPING_COUNT**

**Mode**
READ

**Description**
For AQ namespace. Count of notifications received in the group.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_AQ_NTFN_GROUPING_ MSGID_ARRAY**

**Mode**
READ

**Description**
For AQ namespace. The group: an OCI Collection of message IDs.

**Attribute Data Type**
`OCIColl **`

**OCI_ATTR_CONSUMER_NAME**

**Mode**
READ

**Description**
Consumer name of the notification.

**Attribute Data Type**
`oratext *`

**OCI_ATTR_MSG_PROP**

**Mode**
READ

**Description**
Message properties.

**Attribute Data Type**
`OCIAQMsgProperties **`

**OCI_ATTR_NFY_FLAGS**

**Mode**
READ

**Description**
0 = regular, 1 = timeout notification, 2 = grouping notification.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_NFY_MSGID**

**Mode**
READ

**Description**
Message ID.

**Attribute Data Type**
`OCIRaw *`

**OCI_ATTR_QUEUE_NAME**

**Mode**
READ

**Description**
The queue name of the notification.

**Attribute Data Type**
`oratext *`

## A.22.4 Invalidated Query Attributes

Lists and describes invalidated query attributes.

This section describes `OCI_DTYPE_CQDES` attributes.

> **See Also:**
>
> *Oracle Database Development Guide* for more information about the `OCI_DTYPE_CQDES` continuous query notification descriptor

**OCI_ATTR_CQDES_OPERATION**

**Mode**
READ

**Description**
The operation that occurred on the query. It can be one of two values: `OCI_EVENT_QUERYCHANGE` (query result set change) or `OCI_EVENT_DEREG` (query unregistered).

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_CQDES_QUERYID**

**Mode**
READ

**Description**
Query ID of the query that was invalidated.

**Attribute Data Type**
`ub8 *`

**OCI_ATTR_CQDES_TABLE_CHANGES**

**Mode**
READ

**Description**
A collection of table continuous query descriptors describing DML or DDL operations on tables that caused the query result set change. Each element of the collection is of type `OCI_DTYPE_TABLE_CHDES`.

**Attribute Data Type**
`OCIColl *`

## A.23 Direct Path Loading Handle Attributes

Lists and describes direct path loading handle attributes.

The following attributes are used for the direct path loading handle.

> **See Also:**
>
> Direct Path Loading Overview and Direct Path Loading of Object Types for information about direct path loading and allocating the direct path handles

## A.23.1 Direct Path Context Handle (OCIDirPathCtx) Attributes

Lists and describes direct path context handle (OCIDirPathCtx) attributes.

The following attributes are used for the direct path context handle.

**OCI_ATTR_BUF_SIZE**

**Mode**
READ/WRITE

**Description**
Sets the size of the stream transfer buffer. Default value is 64 KB.

**Attribute Data Type**
`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 the character set ID is not specified at the column level or the table level, then the Global support environment setting is used.

**Attribute Data Type**
`ub2 */ub2 *`

**OCI_ATTR_DATEFORMAT**

**Mode**
READ/WRITE

**Description**
Default date format string for `SQLT_CHR` 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 Data Type**
`oratext **/oratext *`

**OCI_ATTR_DIRPATH_DCACHE_DISABLE**

**Mode**
READ/WRITE

**Description**
Setting this attribute to 1 indicates that the date cache is to be disabled if exceeded. The default value is 0, which means that lookups in the cache continue on cache overflow.

**See Also:**

About Using a Date Cache in Direct Path Loading of Dates in OCI for a complete description of this attribute and of the four following attributes

**Attribute Data Type**
`ub1 */ub1 *`

### OCI_ATTR_DIRPATH_DCACHE_HITS

**Mode**
READ

**Description**
Queries the number of date cache hits.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_DIRPATH_DCACHE_MISSES

**Mode**
READ

**Description**
Queries the current number of date cache misses.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_DIRPATH_DCACHE_NUM

**Mode**
READ

**Description**
Queries the current number of entries in a date cache.

**Attribute Data Type**
`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 this to 0, which is the default value.

**Attribute Data Type**
`ub4 */ub4 *`

### OCI_ATTR_DIRPATH_DEF_EXP_CACHE_SIZE

**Mode**
READ/WRITE

**Description**
Specifies the number of default expressions that are evaluated at a time. The default is 100. For default expressions that must be evaluated for every row, increasing this value may improve performance.
Valid values: `UB4`

**Attribute Data Type**
`ub4 */ub4 *`

**Example**

```
     ub4 default_cache = 200;

     OCIAttrSet ((void *)dpctx,
                      (ub4)OCI_HTYPE_DIRPATH_CTX,
                      (void *)&default_cache,
                      (ub4)0, (ub4)OCI_ATTR_DIRPATH_DEF_EXP_CACHE_SIZE,
errhp);
```

### OCI_ATTR_DIRPATH_DEFAULTS

**Mode**
Read/Write

**Description**
Specifies how the direct path API handles default expressions for columns that are not explicitly being loaded.
Valid values are:

- `OCI_DIRPATH_DEFAULTS_DEFAULT`

  Evaluate once, unless a sequence is involved, then evaluate every row.

  Error if an unsupported default value is seen. This is the default.
- `OCI_DIRPATH_DEFAULTS_EVALUATE_ONCE`

  Evaluate once, at the start of the load. Error if an unsupported default value is seen.
- `OCI_DIRPATH_DEFAULTS_EVALUATE_EVERY_ROW`

  Evaluate every row. Error if an unsupported default value is seen.
- `OCI_DIRPATH_DEFAULTS_IGNORE`

  Ignore all defaults, load NULLs.
- `OCI_DIRPATH_DEFAULTS_IGNORE_UNSUPPORTED_EVALUATE_ONCE`

  Ignore unsupported defaults, load NULLS, evaluate supported once.

- `OCI_DIRPATH_DEFAULTS_IGNORE_UNSUPPORTED_EVALUATE_EVERY_ROW`

  Ignore unsupported defaults, load NULLS, evaluate supported every row.

**Attribute Data Type**
`ub1 */ub1 *`

**Example**

```
     ub1 dirpath_handling = OCI_DIRPATH_DEFAULTS_EVALUATE_EVERY_ROW;

     OCIAttrSet ((void *)dpctx,
                      (ub4)OCI_HTYPE_DIRPATH_CTX,
                      (void *)&dirpath_handling,
                      (ub4)0, (ub4)OCI_ATTR_DIRPATH_DEFAULTS, errhp);
```

### OCI_ATTR_DIRPATH_FLAGS

**Mode**
READ/WRITE

**Description**
Flags used to control behavior of the load.
`OCI_DIRPATH_FLAGS_VLDT` 0x01 – validate format for Oracle `NUMBER` and `DATE` data when the stream is parsed on the server. The default value is to not set this flag because it is an expensive operation. It can be used when you suspect that there is a problem with `OCIDirPath` generating invalid internal representation of dates and numbers.

**Example**

```
ub4 dirpath_flags = OCI_ATTR_DIRPATH_FLAGS_VLDT;

OCIAttrSet ((void *)dpctx,
                  (ub4)OCI_HTYPE_DIRPATH_CTX,
                 (void *)&dirpath_flags,
                 (ub4)0, (ub4)OCI_ATTR_DIRPATH_FLAGS, errhp);
```

**Attribute Data Type**
`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 Data Type**
`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 Data Type**
`ub1 */ub1 *`

### OCI_ATTR_DIRPATH_NO_INDEX_ERRORS

**Mode**
READ/WRITE

**Description**
When `OCI_ATTR_DIRPATH_NO_INDEX_ERRORS` is 1, indexes are not set as unusable at any time during the load. If any index errors are detected, the load is aborted. That is, no rows are loaded, and the indexes are left as is. The default is 0.

**Attribute Data Type**
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 Data Type**
`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((void *)dpctx,
           (ub4)OCI_HTYPE_DIRPATH_CTX,
           (void *) obj_type,
           (ub4)strlen((const char *) obj_type),
           (ub4)OCI_ATTR_DIRPATH_OBJ_CONSTR, errhp);
```

**Attribute Data Type**
`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 Data Type**
`ub1 */ub1 *`

### OCI_ATTR_DIRPATH_PGA_LIM

**Mode**
READ/WRITE

**Description**
The current partition loading memory limit. Once this limit is reached, some partition loading will be delayed to save memory. The value is in KB.

**Attribute Data Type**
`ub4 */ub4 *`

### OCI_ATTR_DIRPATH_REJECT_ROWS_REPCH

**Mode**
READ/WRITE

**Description**
If set to 1, any rows with character conversions that use the replacement character will be rejected.

- 0 - Allow use of the replacement character in conversions.
- 1 - Reject rows if the replacement character was used during conversion.

**Attribute Data Type**
ub1 */ub1 *

### OCI_ATTR_DIRPATH_SKIPINDEX_METHOD

**Mode**
READ/WRITE

**Description**
Indicates how the handling of unusable indexes is 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 Data Type**
`ub1 */ub1 *`

### OCI_ATTR_DIRPATH_SPILL_PASSES

**Mode**
READ

**Description**
The number of passes required to load all partitions. If the direct path PGA limit is exceeded, this will likely be greater than one. Increasing the PGA limit, using the attribute `OCI_ATTR_DIRPATH_PGA_LIM`, can decrease the number of passes, but can also exceed available memory.

**Attribute Data Type**
`ub4 *`

### 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:**
>
> "About Accessing Column Parameter Attributes"

**Attribute Data Type**
`OCIParam* *`

### OCI_ATTR_NAME

**Mode**
READ/WRITE

**Description**
Name of the table to be loaded into.

**Attribute Data Type**
`oratext**/oratext *`

### OCI_ATTR_NUM_COLS

**Mode**
READ/WRITE

**Description**
Number of columns being loaded in the table.

**Attribute Data Type**
`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 Data Type**
`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 Data Type**
`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 that belongs to the table.

**Attribute Data Type**
`oratext **/oratext *`

## A.23.2 Direct Path Function Context Handle (OCIDirPathFuncCtx) Attributes

Lists and describes direct path function context handle (OCIDirPathFuncCtx) attributes.

For further explanations of these attributes, see Direct Path Function Context and Attributes.

**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 nonscalar 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 Data Type**
`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 nonscalar column) is set with the `OCI_ATTR_NUM_COLS` attribute.

> **See Also:**
>
> About Accessing Column Parameter Attributes

**Attribute Data Type**
`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 Data Type**
`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 Data Type**
`ub2 */ub2 *`

**OCI_ATTR_NUM_ROWS**

**Mode**
READ

**Description**
The number of rows loaded so far.

**Attribute Data Type**
`ub4 *`

## A.23.3 Direct Path Function Column Array Handle (OCIDirPathColArray) Attributes

Lists and describes direct path function column array handle (OCIDirPathColArray) attributes.

The following attributes are used for the direct path function column array handle.

**OCI_ATTR_COL_COUNT**

**Mode**
READ

**Description**
Last column of the last row processed.

**Attribute Data Type**
`ub2 *`

**OCI_ATTR_NUM_COLS**

**Mode**
READ

**Description**
Column dimension of the column array.

**Attribute Data Type**
`ub2 *`

**OCI_ATTR_NUM_ROWS**

**Mode**
READ

**Description**
Row dimension of the column array.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_ROW_COUNT**

**Mode**
READ

**Description**
Number of rows successfully converted in the last call to `OCIDirPathColArrayToStream()`.

**Attribute Data Type**
`ub4 *`

## A.23.4 Direct Path Stream Handle (OCIDirPathStream) Attributes

Lists and describes direct path stream handle (OCIDirPathStream) attributes.

The following attributes are used for the direct path stream handle.

**OCI_ATTR_BUF_ADDR**

**Mode**
READ

**Description**
Buffer address of the beginning of the stream data.

**Attribute Data Type**
`ub1 **`

**OCI_ATTR_BUF_SIZE**

**Mode**
READ

**Description**
Size of the stream data in bytes.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_ROW_COUNT**

**Mode**
READ

**Description**
Number of rows successfully loaded by the last `OCIDirPathLoadStream()` call.

**Attribute Data Type**
`ub4 *`

**OCI_ATTR_STREAM_OFFSET**

**Mode**
READ

**Description**
Offset into the stream buffer of the last processed row.

**Attribute Data Type**
`ub4 *`

## A.23.5 Direct Path Column Parameter Attributes

Describes how direct path column parameter attributes are used.

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 nonscalar, 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.

### A.23.5.1 About Accessing Column Parameter Attributes

Lists and describes direct path 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:**
>
> Direct Path Load Examples for Scalar Columns for the data structures defined in the listings

```
...
  /* set number of columns to be loaded */
  OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
            OCIAttrSet((void *)dpctx, (ub4)OCI_HTYPE_DIRPATH_CTX,
                       (void *)&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((void *)dpctx, OCI_HTYPE_DIRPATH_CTX,
                       (void *)&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 void  *)ctlp->colLstDesc_ctl,
                        (ub4)OCI_DTYPE_PARAM, ctlp->errhp_ctl,
                        (void  **)&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((void  *)colDesc, (ub4)OCI_DTYPE_PARAM,
                       (void  *)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((void  *)colDesc, (ub4)OCI_DTYPE_PARAM,
                       (void  *)&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((void  *)colDesc, (ub4)OCI_DTYPE_PARAM,
                       (void  *)&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((void  *)colDesc, (ub4)OCI_DTYPE_PARAM,
                       (void  *)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((void  *)colDesc, (ub4)OCI_DTYPE_PARAM,
                       (void  *)&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((void  *)colDesc, (ub4)OCI_DTYPE_PARAM,
                       (void  *)&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((void  *)colDesc, (ub4)OCI_DTYPE_PARAM,
                       (void  *)&colp->csid_col, (ub4)0,
                       (ub4)OCI_ATTR_CHARSET_ID, ctlp->errhp_ctl));
  }
  /* free the parameter handle to the column descriptor */
  OCI_CHECK((void  *)0, 0, ociret, ctlp,
             OCIDescriptorFree((void  *)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 Data Type**
`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 Data Type**
`ub4 */ub4 *`

### OCI_ATTR_DATA_TYPE

**Mode**
READ/WRITE

**Description**
Returns or sets the external data type of the column. Valid data types are:

- `SQLT_CHR`
- `SQLT_DATE`
- `SQLT_TIMESTAMP`
- `SQLT_TIMESTAMP_TZ`
- `SQLT_TIMESTAMP_LTZ`
- `SQLT_INTERVAL_YM`
- `SQLT_INTERVAL_DS`
- `SQLT_CLOB`
- `SQLT_BLOB`
- `SQLT_INT`
- `SQLT_UIN`
- `SQLT_FLT`
- `SQLT_PDN`

- `SQLT_BIN`
- `SQLT_NUM`
- `SQLT_NTY`
- `SQLT_REF`
- `SQLT_VST`
- `SQLT_VNU`

**Attribute Data Type**
`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 Data Type**
`oratext **/oratext *`

### OCI_ATTR_DIRPATH_OID

**Mode**
READ/WRITE

**Description**
Indicates that the column to load into is an object table's object ID column.

**Attribute Data Type**
`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 Data Type**
`ub1 */ub1 *`

### OCI_ATTR_NAME

**Mode**
READ/WRITE

**Description**
Returns or sets the name of the column that is being loaded. Initialize both the column name and the column name length to 0 before calling `OCIAttrGet()`.

**Attribute Data Type**
`oratext **/oratext *`

**OCI_ATTR_PRECISION**

**Mode**
READ/WRITE

**Description**
Returns or sets the precision.

**Attribute Data Type**
`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 data types.

**Attribute Data Type**
`sb1 */sb1 *`

## A.24 Process Handle Attributes

Lists and describes 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`.

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.

**See Also:**

OCI_ATTR_SHARED_HEAPALLOC

### OCI_ATTR_MEMPOOL_APPNAME

**Mode**
READ/WRITE

**Description**
Executable name or fully qualified path name of the executable.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_MEMPOOL_HOMENAME

**Mode**
READ/WRITE

**Description**
Directory name where the executables that use the same shared subsystem instance are located.

**Attribute Data Type**
`oratext *`

### OCI_ATTR_MEMPOOL_INSTNAME

**Mode**
READ/WRITE

**Description**
Any user-defined name to identify an instance of the shared subsystem.

**Attribute Data Type**
`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((void  *)0, (ub4) OCI_HTYPE_PROC,
           (void  *)&plsz, (ub4) 0, (ub4) OCI_ATTR_POOL_SIZE, 0);
```

**Attribute Data Type**
`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 AND'ed with that mode. For example:

```
ub4 mode;
boolean is_shared;

OCIAttrGet((void  *)0, (ub4)OCI_HTYPE_PROC,
           (void  *) &mode, (ub4 *) 0,
           (ub4)OCI_ATTR_PROC_MODE, 0);

is_shared = mode & OCI_SHARED;
```

**Attribute Data Type**
`ub4 *`

## A.25 Event Handle Attributes

Lists and describes event handle attributes.

The `OCIEvent` handle encapsulates the attributes from the event payload. This handle is implicitly allocated before the event callback is called.

The event callback obtains the attributes of an event using `OCIAttrGet()` with the following attributes.

**See Also:**

HA Event Notification

**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. This is also a server handle attribute.

**Attribute Data Type**
`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. This is also a server handle attribute.

**Attribute Data Type**
`oratext **`

### OCI_ATTR_EVENTTYPE

**Mode**
READ

**Description**
The type of event that occurred, `OCI_EVENTTYPE_HA`.

**Attribute Data Type**
`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 Data Type**
`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 an Oracle Real Application Clusters (Oracle RAC) HA DOWN event.

**Attribute Data Type**
`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 an Oracle RAC HA DOWN event.

**Attribute Data Type**
`OCIServer **`

### OCI_ATTR_HA_STATUS

**Mode**
READ

**Description**
Valid value is `OCI_HA_STATUS_DOWN`. Only `DOWN` events are subscribed to currently.

**Attribute Data Type**
`ub4 *`

### OCI_ATTR_HA_TIMESTAMP

**Mode**
READ

**Description**
The time that the HA event occurred.

**Attribute Data Type**
`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 Data Type**
`oratext **`

### OCI_ATTR_INSTNAME

**Mode**
READ

**Description**
When called as an event handle attribute, `OCIAttrGet()` retrieves the name of the instance that has been affected by this event. This is also a server handle attribute. This attribute can also be used on service context to return the shard instance name

for a given connection. This attribute can be used to find the shard instance name out of a shard instance descriptor that is filed by an `OCIShardInstancesGet()` call.

**Attribute Data Type**
`oratext **`

**Example**
The following example shows for a given connection (svchp) how to get the shard instance name.

```
Oratext instanceName [OCI_INSTNAME_MAXLEN];
ub4     instanceNameLen;
   OCIATTRGet(svchp,
              OCI_HTYPE_SVCCTX,
              instanceName,
              (ub4 *) &instanceNameLen,
              OCI_ATTR_INSTNAME,
              errhp);
```

### OCI_ATTR_INSTSTARTTIME

**Mode**
READ

**Description**
When called with this attribute, `OCIAttrGet()` retrieves the start time of the instance that has been affected by this event. This is also a server handle attribute.

**Attribute Data Type**
`OCIDateTime **`

### 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. The name length is `ub4 *`. This is also a server handle attribute.

**Attribute Data Type**
`oratext **`

# B

# OCI Demonstration Programs

Lists example programs with sample code demonstrating the allocation and use of OCI handles.

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.

You must install the demonstration programs as described in *Oracle Database Examples Installation Guide*. The location, names, and availability of the programs may vary on different operating systems. On a Linux or UNIX workstation, the programs are installed in the `$ORACLE_HOME/rdbms/demo` directory. For Windows systems, see Getting Started with OCI for Windows.

OCI header files that are required for OCI client application development on Linux or 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. On Windows systems, `make.bat` is the analogous file in the samples directory. There are instructions in the makefiles.

Unless you significantly modify the `demo_rdbms.mk` file, you are not affected by changes you make as long the `demo_rdbms.mk` file 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 results in easier maintenance of your own makefiles.

When a specific header or SQL file is required by the application, these files are also included as specified in the demonstration program file. 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. Look for related files with the `.sql` extension.

**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`<br>`cdemodr2.c`<br>`cdemodr3.c` | Using `INSERT`, `UPDATE`, and `DELETE` statements, with `RETURNING` clause used with basic data types, LOBs and REFs |

**Table B-1    (Cont.) OCI Demonstration Programs**

| Program Name | Features Illustrated |
|---|---|
| `cdemodsa.c` | Describing information about a table |
| `cdemodsc.c` | Describing information about an object type |
| `cdemofo.c` | Registering and operating application failover callbacks |
| `cdemolb.c` | Creating and inserting LOB data and then reading, writing, coping, appending, and trimming the data |
| `cdemolb2.c` | Writing and reading of `CLOB` and `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` | Inserting and selecting 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 Database Advanced Queuing User's Guide*) |
| `ociaqdemo00.c` | Database 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`,<br>`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`,<br>`cdemoucbl.c` | Using static and dynamic user callbacks |
| `cdemoupk.c`,<br>`cdemoup1.c`,<br>`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) |

**Table B-1 (Cont.) OCI Demonstration Programs**

| Program Name | Features Illustrated |
|---|---|
| `cdemoin1.c` | Inheritance demo that 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 |
| `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 |
| `cdemoqc.c` | Query caching using SQL hints |
| `cdemoqc2.c` | Query caching using SQL hints and table annotation |
| `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 code point feature |

# C

# OCI Function Server Round-Trips

This appendix provides information about server round-trips incurred during various OCI calls.

This information can help programmers to determine the most efficient way to accomplish a particular task in an application. A **server round-trip** is defined as the trip from the client to the server and back to the client.

When using client-side SQL, OCI provides:

- Enhanced array data manipulation language (DML) capability for arrays
- Ability to associate a commit request with a run to reduce round-trips
- Optimization for queries using transparent prefetch buffers to reduce round-trips

As is often the case, there are tradeoffs. For example, for queries the trade off is often arraysize and prefetch size versus the number of underlying fetches.

> **See Also:**
>
> - Using SQL Statements in OCI for an overview and more specific information on tips to reduce server round-trips
> - OCI Array Interface to significantly reduce round-trips to the database when you are updating or inserting a large volume of data
> - About Setting Prefetch Count to minimize server round-trips

.

This appendix contains these topics:

- Relational Function Round-Trips
- LOB Function Round-Trips
- Object and Cache Function Round-Trips
- Describe Operation Round-Trips
- Data Type Mapping and Manipulation Function Round-Trips
- Any Type and Data Function Round-Trips
- Other Local Functions

## C.1 Relational Function Round-Trips

Describes for each OCI relational function the number of server round-trips.

Table C-1 lists the number of server round-trips required by each OCI relational function.

**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 |
| `OCILdaToSvcCtx()` | 0 |
| `OCILogoff()` | 1 |
| `OCILogon()` | 1 |
| `OCILogon2()` | Connection pool or session pool: same as OCISessionGet()<br>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.<br>Normal: 1 login |
| `OCISessionPoolCreate()` | sessMin * logins |
| `OCISessionPoolDestroy()` | Sessions in cache * logoffs |
| `OCISessionRelease()` | Session pooling: 0, except when explicit session destroys flag set<br>Normal: 1 login |

**Table C-1 (Cont.) Server Round-Trips for Relational Operations**

| Function | Number of Server Round-Trips |
|---|---|
| `OCIStmtExecute()` | 1 |
| `OCIStmtFetch()` | 0 or 1 |
| `OCIStmtFetch2()` | 0 in prefetch, otherwise 1 |
| `OCIStmtGetPieceInfo()` | 1 |
| `OCIStmtPrepare()` | 0 |
| `OCIStmtPrepare2()` | 0 |
| `OCIStmtSetPieceInfo()` | 0 |
| `OCISvcCtxToLda()` | 0 |
| `OCITerminate()` | 1 |
| `OCITransCommit()` | 1 |
| `OCITransDetach()` | 1 |
| `OCITransForget()` | 1 |
| `OCITransPrepare()` | 1 |
| `OCItransRollback()` | 1 |
| `OCITransStart()` | 1 |
| `OCIUserCallbackGet()` | 0 |
| `OCIUserCallbackRegister()` | 0 |

## C.2 LOB Function Round-Trips

Describes for each OCI LOB function the number of server 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.

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 |

**Table C-2 (Cont.) Server Round-Trips for OCILob Calls**

| Function | Number of Server Round-Trips |
|---|---|
| `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 |
| `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 |

**Table C-2 (Cont.) Server Round-Trips for OCILob Calls**

| Function | Number of Server Round-Trips |
|---|---|
| `OCILobRead2()` | 0 or 1 |
| `OCILobTrim()` | 1 |
| `OCILobTrim2()` | 1 |
| `OCILobWrite()` | 0 or 1 |
| `OCILobWrite2()` | 0 or 1 |
| `OCILobWriteAppend()` | 0 or 1 |
| `OCILobWriteAppend2()` | 0 or 1 |

**See Also:**

- About Binding LOB Data for usage and examples for both `INSERT` and `UPDATE` statements
- About Defining LOB Data
  for usage and examples of `SELECT` statements

## C.3 Object and Cache Function Round-Trips

Describes for each OCI object and cache function the number of server 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 |
|---|---|
| `OCICacheFlush()` | 1 |
| `OCICacheFree()` | 0 |
| `OCICacheRefresh()` | 1 |
| `OCICacheUnmark()` | 0 |
| `OCICacheUnpin()` | 0 |
| `OCIObjectArrayPin()` | 1 |
| `OCIObjectCopy()` | 0 |
| `OCIObjectExists()` | 0 |
| `OCIObjectFlush()` | 1 |
| `OCIObjectFree()` | 0 |
| `OCIObjectGetInd()` | 0 |

**Table C-3 (Cont.) Server Round-Trips for Object and Cache Functions**

| Function | Number of Server Round-Trips |
|---|---|
| `OCIObjectGetObjectRef()` | 0 |
| `OCIObjectGetTypeRef()` | 0 |
| `OCIObjectIsDirty()` | 0 |
| `OCIObjectIsLocked()` | 0 |
| `OCIObjectLock()` | 1 |
| `OCIObjectMarkDelete()` | 0 |
| `OCIObjectMarkDeleteByRef()` | 0 |
| `OCIObjectMarkUpdate()` | 0 |
| `OCIObjectNew()` | 0 |
| `OCIObjectPin()` | 1; 0 if the desired object is already in cache |
| `OCIObjectPinCountReset()` | 0 |
| `OCIObjectPinTable()` | 1 |
| `OCIObjectRefresh()` | 1 |
| `OCIObjectUnmark()` | 0 |
| `OCIObjectUnmarkByRef()` | 0 |
| `OCIObjectUnpin()` | 0 |

## C.4 Describe Operation Round-Trips

Describes for each OCI operation function the number of server round-trips.

Table C-4 lists the number of server round-trips required by `OCIDescribeAny()`, `OCIAttrGet()`, and `OCIParamGet()`.

**Table C-4 Server Round-Trips for Describe Operations**

| Function | Number of Server Round-Trips |
|---|---|
| `OCIAttrGet()` | 2 round-trips to describe a type if the type objects are not in the object cache<br>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.<br>0 if all the type objects to be described are already in the object cache following the first `OCIAttrGet()` call. |
| `OCIDescribeAny()` | 1 round-trip to get the REF of the type descriptor object |
| `OCIParamGet()` | 0 |

## C.5 Data Type Mapping and Manipulation Function Round-Trips

Describes for each OCI data type mapping and manipulation function the number of server round-trips.

Table C-5 lists the number of round-trips for the data type mapping and manipulation functions. 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 Data Type Manipulation Functions**

| Function | Number of Server Round-Trips |
|---|---|
| `OCIColl*()` | 0; 1 if the collection is not loaded in the cache |
| `OCIDate*()` | 0 |
| `OCIIter*()` | 0; 1 if the collection is not loaded in the cache |
| `OCINumber*()` | 0 |
| `OCIRaw*()` | 0 |
| `OCIRef*()` | 0 |
| `OCIString*()` | 0 |
| `OCITable*()` | 0; 1 if the nested table is not loaded in the cache |

## C.6 Any Type and Data Function Round-Trips

Describes for each OCI Any Type and Data function the number of server round-trips.

Table C-6 lists the number of server round-trips required by Any Type and Data functions. 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 |

## C.7 Other Local Functions

Describes for each OCI other local function the number of server round-trips.

Table C-7 lists the functions that 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 makes one round-trip to fetch the type descriptor object. |
| `OCIAttrSet()` | |
| `OCIBindArrayOfStruct()` | |
| `OCIDefineArrayOfStruct()` | |
| `OCIBindByName()` | |
| `OCIBindByPos()` | |
| `OCIBindDynamic()` | |
| `OCIBindObject()` | |
| `OCIDefineByPos()` | |
| `OCIDefineDynamic()` | |
| `OCIDefineObject()` | |
| `OCIDescriptorAlloc()` | |
| `OCIDescriptorFree()` | |
| `OCIEnvCreate()` | |
| `OCIEnvInit()` | |
| `OCIErrorGet()` | |
| `OCIHandleAlloc()` | |
| `OCIHandleFree()` | |
| `OCILdaToSvcCtx()` | |
| `OCIStmtGetBindInfo()` | |
| `OCIStmtPrepare()` | |
| `OCIStmtRelease()` | |
| `OCIStmtPrepare2()` | |
| `OCISvcCtxToLda()` | |

D

# Getting Started with OCI for Windows

How to get 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?
- Oracle Directory Structure for Windows
- Sample OCI Programs for Windows
- About Compiling OCI Applications for Windows
- About Linking OCI Applications for Windows
- About Running OCI Applications for Windows
- Oracle XA Library
- About Using the Object Type Translator for Windows

## D.1 What Is Included in the OCI Package for Windows?

The OCI for Windows package includes the additional libraries required for linking your OCI programs.

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

**See Also:**

About Oracle Instant Client for a simplified OCI installation option

## D.2 Oracle Directory Structure for Windows

OCI is included in the default Oracle Database installation.

When you install Oracle Database, Oracle Universal Installer creates the OCI files in the `oci`, `bin`, and `precomp` directories under the *`ORACLE_BASE`*`\`*`ORACLE_HOME`* directory. These files include the library files needed to link and run OCI applications, and link with other Oracle for Microsoft Windows products, such as Oracle Forms.

The *`ORACLE_BASE`*`\`*`ORACLE_HOME`* directory contains the following directories described in Table D-1 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 |

## D.3 Sample OCI Programs for Windows

Where sample program files and corresponding project files are copied when you install OCI.

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.

The `ociucb.c` program 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`.

## D.4 About 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++ 8.0, you must put in the appropriate path, `\oracle\db_1\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 7.1 or later.

> **See Also:**
>
> Your compiler's documentation for specific information about compiling your application and special compiler options

# D.5 About 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 relinking or compilation with every release.

When using `oci.lib` with the Microsoft compiler, you do not have to indicate any special link options.

This section includes the following topics:

- oci.lib
- Client DLL Loading When Using Load Library()

## D.5.1 oci.lib

Why you need applications linked with oci.lib.

Oracle recommends that applications be linked with `oci.lib`, which takes care of loading the correct versions of the Oracle DLLs.

## D.5.2 Client DLL Loading When Using Load Library()

How directories are searched when using the LoadLibrary.

The following directories are searched in this order by the `LoadLibrary()` function for client DLL loading:

- Directory from which the application is loaded or the directory where `oci.dll` is located
- Current directory
- Windows:
  - The 32-bit Windows system directory (`system32`). Use the `GetWindowsDirectory()` function of the Windows API to obtain the path of this directory.
  - The 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

## D.6 About Running OCI Applications for Windows

What’s needed to tun an 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.

## D.7 Oracle XA Library

The XA application programming interface (API) is typically used to enable an Oracle Database to interact with a transaction processing (TP) monitor, such as:

- Oracle 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 Oracle Database Enterprise Edition. Table D-2 lists the components created in your Oracle home directory. The `oci.lib` import library contains the XA exports.

**Table D-2    Oracle XA Components**

| Component | Location |
|---|---|
| `xa.h` | `*ORACLE_BASE*\*ORACLE_HOME*\oci\include` |

This section includes the following topics:

- About Compiling and Linking an OCI Program with the Oracle XA Library
- About Using XA Dynamic Registration
- XA and TP Monitor Information

## D.7.1 About Compiling and Linking an OCI Program with the Oracle XA Library

How to compile and link an OCI program with the Oracle XA Library.

To compile and link an OCI program with the Oracle XA Library:

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.
2. Link *`program`*`.obj` with the libraries shown in Table D-3:

**Table D-3 Link Libraries**

| Library | Location |
|---|---|
| `oraxa12.lib` | *`ORACLE_BASE`*`\`*`ORACLE_HOME`*`\rdbms\xa` |
| `oci.lib` | *`ORACLE_BASE`*`\`*`ORACLE_HOME`*`\oci\lib\msvc` or,<br>*`ORACLE_BASE`*`\`*`ORACLE_HOME`*`\oci\lib\bc` |

3. Run *`program`*`.exe`.

## D.7.2 About Using XA Dynamic Registration

The database supports the use of XA dynamic registration.

XA dynamic registration improves the performance of applications interacting 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 systems on which your TP monitor is running. See either of the following sections for instructions:

- Adding an Environmental Variable for the Current Session
- About Adding a Registry Variable for All Sessions
- Adding a Registry Variable:

### D.7.2.1 Adding an Environmental Variable for the Current Session

Adding an environmental variable at the command prompt affects only the current session.

**To Add 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
```

In this example, *`vendor`*`.dll` is the TP monitor DLL provided by your vendor.

### D.7.2.2 About Adding a Registry Variable for All Sessions

Adding a registry variable affects all sessions on your Windows system.

This is useful for computers where only one TP monitor is running.

### D.7.2.3 Adding a Registry Variable:

To add a registry variable for the TP monitor provided by your vendor, follow these steps.

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. Select the **Add Value** 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.
7. Click **OK**. The String Editor dialog box appears.
8. Enter *`vendor`*`.dll` in the **String** field, where *`vendor`*`.dll` is the TP monitor DLL provided by your vendor.
9. Click **OK**. The Registry Editor adds the parameter.
10. Select **Exit** from the **Registry** menu.

    The registry exits.

## D.7.3 XA and TP Monitor Information

See the *Distributed TP: The XA Specification* and other specific TP monitor and XA information.

*Distributed TP: The XA Specification* (C193) published by the Open Group at the web site: Distributed TP: The XA Specification

**See Also:**

- *Oracle Database Development Guide*, the topic about developing applications with Oracle XA, for more information about the Oracle XA Library and using XA dynamic registration
- Oracle Tuxedo for more information

# D.8 About Using the Object Type Translator for Windows

To take advantage of objects, run the Object Type Translator (OTT) against the database to generate a header file 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 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 10*g* 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. A configuration file can also be named on the command line. For Windows, the configuration file is `ottcfg.cfg`, located in *`ORACLE_BASE`*`\`*`ORACLE_HOME`*`\precomp\admin`.

## E

# Deprecated OCI Features and Functions

Deprecated OCI features and functions are those that may become obsolete in a future release. Replacement OCI functions are mentioned where applicable.

Beginning with Oracle Database release 18c, version 18.1, the result set descriptor `OCI_DTYPE_RSET`.

OCI release 7.3 API beginning with Oracle Database 12*c* Release 1 (12.1).

Table E-1 lists the OCI functions that were deprecated in releases previous to Oracle 11*g* R2 (11.2) unless otherwise noted. In a future release, these functions may become obsolete.

**Table E-1 Deprecated OCI Functions**

| Function Group | Deprecated Functions |
|---|---|
| Initialize | OCIEnvInit(), OCIInitialize() |
| Statement | OCIStmtFetch()<br>Beginning with Oracle Database 12*c* Release 2 (12.2): OCIStmtPrepare() |
| Lob | OCILobCopy(), OCILobErase(), OCILobGetLength(), OCILobLoadFromFile(), OCILobRead(), OCILobTrim(), OCILobWrite(), OCILobWriteAppend()<br>Beginning with Oracle Database 12*c* Release 2 (12.2): OCILobDisableBuffering(),OCILobEnableBuffering(), OCILobFlushBuffer() |
| Database Advanced Queuing functions | OCIAQListen() |

This section includes the following topics:

- Deprecated Initialize Functions
- Deprecated Statement Functions
- Deprecated Lob Functions
- Deprecated Database Advanced Queuing Functions

## E.1 Deprecated Initialize Functions

Describes the deprecated initialize functions.

Table E-2 lists the deprecated Initialize functions that are described in this section.

**Table E-2 Deprecated Initialize Functions**

| Function | Purpose |
|---|---|
| OCIEnvInit() | Initialize an environment handle. |
| OCIInitialize() | Initialize OCI process environment. |

## E.1.1 OCIEnvInit()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

### Purpose

Allocates and initializes an OCI environment handle. This function is deprecated.

### Syntax

```
sword OCIEnvInit ( OCIEnv    **envhpp,
                   ub4         mode,
                   size_t      xtramemsz,
                   void      **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_UCB`

In `OCI_DEFAULT` mode, the OCI library always mutexes handles.

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

**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:**
>
> Use OCIEnvCreate() instead of the `OCIInitialize()` and `OCIEnvInit()` calls. `OCIInitialize()` and `OCIEnvInit()` calls are supported for backward compatibility.

This call allocates and initializes an OCI environment handle. No changes are made to an initialized handle. If `OCI_ERROR` or `OCI_SUCCESS_WITH_INFO` is returned, you can use the environment handle 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()`.

### Related Topics

- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIHandleFree()
  Explicitly deallocates a handle
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCITerminate()
  Detaches the process from the shared memory subsystem and releases the shared memory.

> **See Also:**
>
> User Memory Allocation for more information about the `xtramemsz` parameter and user memory allocation

## E.1.2 OCIInitialize()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

### Purpose

Initializes the OCI process environment. This function is deprecated.

### Syntax

```
sword OCIInitialize ( ub4            mode,
                      const void    *ctxp,
                      const void    *(*malocfp)
                                     ( void   *ctxp,
                                        size_t size ),
const void    *(*ralocfp)
               ( void  *ctxp,
                 void  *memptr,
                 size_t newsize ),
const void    (*mfreefp)
               ( void  *ctxp,
                 void  *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` - Uses object features.
- `OCI_EVENTS` - Uses publish-subscribe notifications.

**ctxp (IN)**
User-defined context for the memory callback 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 reallocation 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:**

Use OCIEnvCreate() instead of the deprecated OCIInitialize() call. The `OCIInitialize()` call is 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 reallocation, 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:**

- Overview of OCI Multithreaded Development for information about using the OCI to write multithreaded applications
- OCI Object-Relational Programming for information about OCI programming with objects

### 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, (void  *)0,
     (void  * (*)()) 0, (void  * (*)()) 0,  (void (*)()) 0 );
```

### Related Topics

- OCIHandleAlloc()
  Returns a pointer to an allocated and initialized handle.
- OCIHandleFree()
  Explicitly deallocates a handle
- OCIEnvCreate()
  Creates and initializes an environment handle for OCI functions to work under.
- OCITerminate()
  Detaches the process from the shared memory subsystem and releases the shared memory.

# E.2 Deprecated Statement Functions

Describes the deprecated statement functions.

Table E-3 lists the deprecated Statement functions that are described in this section.

**Table E-3 Deprecated Statement Functions**

| Function | Purpose |
|---|---|
| OCIStmtFetch() | Fetch rows from a query. |
| OCIStmtPrepare() | Prepares a SQL or PL/SQL statement for execution. |

## E.2.1 OCIStmtFetch()

This function was deprecated in a release previous to Oracle 11*g* R2 (11.2).

### Purpose

Fetches rows from a query. This function is deprecated. Use OCIStmtFetch2().

### 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 that you can pass to OCIErrorGet() for diagnostic information when there is an error.

**nrows (IN)**
Number of rows to be fetched from the current position.

**orientation (IN)**
Before 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 these errors occurs:

- 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 Topics**

- OCIStmtFetch2()
  Fetches a row from the (scrollable) result set.

## E.2.2 OCIStmtPrepare()

This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).

### Purpose

Prepares a SQL or PL/SQL statement for execution. This function is deprecated. Use OCIStmtPrepare2().

### Syntax

```
sword OCIStmtPrepare ( OCIStmt       *stmtp,
                       OCIError      *errhp,
                       const OraText *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 that you can pass to OCIErrorGet() for diagnostic information when there is 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 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 as follows:

- `OCI_V7_SYNTAX` - V7 ORACLE parsing syntax.
- `OCI_NTV_SYNTAX` - Syntax depends upon the version of the server.

  `OCI_FOREIGN_SYNTAX` - Specifies the statement to be translated according to the SQL translation profile set in the session.

**mode (IN)**
Similar to the `mode` in the OCIEnvCreate() call, but this one has higher priority because it can override the "naturally" inherited mode setting.
The possible values are `OCI_DEFAULT` (default mode) or `OCI_PREP2_IMPL_RESULTS_CLIENT`. The mode should be passed as `OCI_PREP2_IMPL_RESULTS_CLIENT` when this call is made in an external procedure and implicit results need to be processed. See OCI Support for Implicit Results for more details. 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 code points or in number of bytes, depending on the encoding.

Although 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 are stored in a bind handle that uses settings in this statement handle as the default.

This call does not create an association between this statement handle and any particular server.

Before reexecuting a DDL statement, call this function a second time.

This call does not support Application Continuity and if this call is used, it returns the following error: `Error - ORA-25412: transaction replay disabled by call to OCIStmtPrepare`. Use the `OCIStmtPrepare2()` call to support the use of Application Continuity in an HA infrastructure.

### Related Topics

- OCIStmtPrepare2()
  Prepares a SQL or PL/SQL statement for execution.

**See Also:**

About Preparing Statements for more information about using this call

## E.3 Deprecated Lob Functions

Describes the deprecated LOB functions.

Table E-4 lists the deprecated LOB functions that are described in this section.

Beginning with Oracle Database 12c Release 2 (12.2), most LOB functions including some deprecated LOB functions as indicated in the following table support Application Continuity (see When Application Continuity in OCI Can Fail Over for a complete list).

**Table E-4    Deprecated LOB Functions**

| Function | Purpose |
|---|---|
| OCILobCopy() | Copy all or part of one LOB to another. Supports Application Continuity. |
| OCILobDisableBuffering() | Disables LOB buffering for the input locator. |
| OCILobEnableBuffering() | Enables LOB buffering for the input locator. |
| OCILobErase() | Erase a portion of a LOB. Supports Application Continuity. |
| OCILobFlushBuffer() | Flushes or write s all buffers for this LOB to the server. |
| OCILobGetLength() | Get length of a LOB. |
| OCILobLoadFromFile() | Load a LOB from a `BFILE`. Supports Application Continuity. |
| OCILobRead() | Read a portion of a LOB. Supports Application Continuity. |
| OCILobTrim() | Truncate a LOB. Supports Application Continuity. |
| OCILobWrite() | Write into a LOB. Supports Application Continuity. |
| OCILobWriteAppend() | Write data beginning at the end of a LOB. |

### E.3.1 OCILobCopy()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

**Purpose**

Copies all or a portion of a LOB value into another LOB value. This function is deprecated. Use OCILobCopy2().

### Syntax

```
sword OCILobCopy ( OCISvcCtx        *svchp,
                   OCIError         *errhp,
                   OCILobLocator    *dst_locp,
                   OCILobLocator    *src_locp,
                   ub4               amount,
                   ub4               dst_offset,
                   ub4               src_offset );
```

### Parameters

### Related Topics

- OCILobCopy2()
  Copies all or a portion of a LOB value into another LOB value.

## E.3.2 OCILobDisableBuffering()

This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).

### Purpose

Disables LOB buffering for the input locator. This function is deprecated.

### Syntax

```
sword OCILobDisableBuffering ( OCISvcCtx      *svchp,
                               OCIError       *errhp,
                               OCILobLocator  *locp );
```

### Parameters

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to OCIErrorGet() or diagnostic information when there is 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.

`OCILobDisableBuffering()` throws an error when a remote locator is passed to it.

This function does not accept a `BFILE` locator.

**Related Topics**

- OCILobEnableBuffering()
  This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- OCILobFlushBuffer()
  This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).

## E.3.3 OCILobEnableBuffering()

This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).

### Purpose

Enables LOB buffering for the input locator. This function is deprecated.

### Syntax

```
sword OCILobEnableBuffering ( OCISvcCtx      *svchp,
                              OCIError       *errhp,
                              OCILobLocator  *locp );
```

### Parameters

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to OCIErrorGet() for diagnostic information when there is 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.

`OCILobEnableBuffering()` throws an error when a remote locator is passed to it.

If LOB buffering is enabled for a locator and that locator is passed to one of these routines, an error is returned: OCILobAppend(), OCILobCopy() (deprecated), OCILobCopy2(), OCILobErase() (deprecated), OCILobErase2(), OCILobGetLength() (deprecated), OCILobGetLength2(), OCILobLoadFromFile() (deprecated), OCILobLoadFromFile2(), OCILobTrim() (deprecated), OCILobTrim2(), OCILobWriteAppend() (deprecated), or OCILobWriteAppend2().

This function does not accept a `BFILE` locator.

**Related Topics**

- OCILobDisableBuffering()
  This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- OCILobWrite()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobRead()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCILobRead2()
  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.
- OCILobFlushBuffer()
  This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).
- OCILobWriteAppend()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.

## E.3.4 OCILobErase()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

**Purpose**

Erases a specified portion of the internal LOB data starting at a specified offset. This function is deprecated. Use `OCILobErase2()`.

**Syntax**

```
sword OCILobErase ( OCISvcCtx       *svchp,
                    OCIError        *errhp,
                    OCILobLocator   *locp,
                    ub4             *amount,
                    ub4              offset );
```

**Parameters**

**Related Topics**

- OCILobErase2()
  Erases a specified portion of the internal LOB data starting at a specified offset.

## E.3.5 OCILobFlushBuffer()

This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).

### Purpose

Flushes or write s all buffers for this LOB to the server. This function is deprecated.

### 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 that you can pass to `OCIErrorGet()` for diagnostic information when there is an error.

**locp (IN/OUT)**
An internal locator uniquely referencing the LOB.

**flag (IN)**
When this flag is set to `OCI_LOB_BUFFER_FREE`, the buffer resources for the LOB are freed after the flush. See the 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 actually writes the data in the buffer to the LOB in the database. LOB buffering must have been enabled for the input LOB locator.

`OCILobFlushBuffer()` throws an error when a remote locator is passed to it.

The flush operation, by default, does not free the buffer resources for reallocation to another buffered LOB operation. 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 there is an added cost for acquiring buffer resources and initializing the buffer with the data read from the LOB. This option is intended for client environments that have low on-board memory.

**Related Topics**

- OCILobEnableBuffering()
  This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).
- OCIErrorGet()
  Returns an error message and an Oracle Database error code.
- OCILobWrite()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.
- OCILobRead()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCILobRead2()
  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.
- OCILobDisableBuffering()
  This function was deprecated beginning with Oracle Database 12*c* Release 2 (12.2).
- OCILobWriteAppend()
  This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).
- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.

## E.3.6 OCILobGetLength()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

**Purpose**

Gets the length of a LOB. This function is deprecated. Use `OCILobGetLength2()`.

**Syntax**

```
sword OCILobGetLength ( OCISvcCtx      *svchp,
                        OCIError       *errhp,
                        OCILobLocator  *locp,
                        ub4            *lenp );
```

**Parameters**

**Related Topics**

- OCILobGetLength2()
  Gets the length of a LOB. This function must be used for LOBs of size greater than 4 GB.

## E.3.7 OCILobLoadFromFile()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

### Purpose

Loads and copies all or a portion of the file into an internal LOB. This function is deprecated. Use `OCILobLoadFromFile2()`.

### Syntax

```
sword OCILobLoadFromFile ( OCISvcCtx       *svchp,
                           OCIError        *errhp,
                           OCILobLocator   *dst_locp,
                           OCILobLocator   *src_locp,
                           ub4              amount,
                           ub4              dst_offset,
                           ub4              src_offset );
```

### Parameters

### Related Topics

- OCILobLoadFromFile2()
  Loads and copies all or a portion of the file into an internal LOB. This function must be used for LOBs of size greater than 4 GB.

## E.3.8 OCILobRead()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

### Purpose

Reads a portion of a LOB or `BFILE`, as specified by the call, into a buffer. This function is deprecated. Use `OCILobRead2()`.

### Syntax

```
sword OCILobRead ( OCISvcCtx          *svchp,
                   OCIError           *errhp,
                   OCILobLocator      *locp,
                   ub4                *amtp,
                   ub4                 offset,
                   void               *bufp,
                   ub4                 bufl,
                   void               *ctxp,
                   OCICallbackLobRead (cbfp)
                                      ( void          *ctxp,
                                        const void    *bufp,
                                        ub4            len,
                                        ub1            piece
                                      )
                   ub2                 csid,
                   ub1                 csfrm );
```

**Parameters**

**svchp (IN/OUT)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to OCIErrorGet() for diagnostic information when there is an error.

**locp (IN)**
A LOB or `BFILE` locator that uniquely references the LOB or `BFILE`. This locator must have been 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 Table E-5.

### Table E-5 Characters or Bytes in amtp for OCILobRead()

| LOB or BFILE | Input | Output with Fixed-Width Client-Side Character Set | Output with Varying-Width Client-Side Character Set |
|---|---|---|---|
| `BLOB`s and `BFILE`s | bytes | bytes | bytes |
| `CLOB`s and `NCLOB`s | characters | characters | bytes [1] |

[1] The input amount refers to the number of characters to be read from the server-side `CLOB` or `NCLOB`. The output amount indicates how many bytes were read into the buffer `bufp`.

The parameter `amtp` is the total amount of data read if:

- Data is not read in streamed mode (only one piece is read and there is no polling or callback)
- Data is read in streamed mode with a callback

The parameter `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 is 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, the `amtp` parameter always contains the length of the piece just read.

If a callback function is defined, then this callback function is invoked each time `bufl` bytes are read off the pipe. Each piece is written into `bufp`.

If the callback function is not defined, then the `OCI_NEED_DATA` error code is returned. The application must call `OCILobRead()` over and over again to read more pieces of the LOB until the `OCI_NEED_DATA` error code is not returned. The buffer pointer and the length can be different in each call if the pieces are being read into different sizes and locations.

**offset (IN)**
On input, this is the absolute offset from the beginning of the LOB value. For character LOBs (`CLOB`s, `NCLOB`s) it is the number of characters from the beginning of the LOB, for binary LOBs or `BFILE`s it is the number of bytes. The first position is 1.
If you use streaming (by polling or a callback), specify the offset in the first call; in subsequent polling calls, the offset parameter is ignored. When you use a callback, there is no offset parameter.

**bufp (IN/OUT)**
The pointer to a buffer into which the piece is read. The length of the allocated memory is assumed to be `bufl`.

**bufl (IN)**
The length of the buffer in octets. This value differs from the `amtp` value for `CLOB`s and for `NCLOB`s (`csfrm=SQLCS_NCHAR`) when the `amtp` parameter is specified in terms of characters, and 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 can be registered to be called for each piece. If this is `NULL`, then `OCI_NEED_DATA` is 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.
The `csfrm` parameter 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 you read or write LOBs, specify a character set form (`csfrm`) that matches the form of the locator itself.

For `BFILE`s, the operating system file must exist on the server, and it must have been opened by OCILobFileOpen() or OCILobOpen() using the input locator. Oracle Database must have permission to read the operating system file, and the user must have read permission on the directory object.

When you use the polling mode for `OCILobRead()`, the first call must specify values for `offset` and `amtp`, but on subsequent polling calls to `OCILobRead()`, you 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 `CLOB`s and `NCLOB`s:

- When you use 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 you use 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 cannot be filled with data.

The following applies to client-side fixed-width character sets and server-side varying-width character sets for `CLOB`s and `NCLOB`s:

- 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 you use 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.

**Related Topics**

- OCILobRead2()
  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.

> **See Also:**
>
> - PL/SQL REF CURSORs and Nested Tables in OCI for additional information about Unicode format
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for a description of `BFILE`s
> - The demonstration programs included with your Oracle Database installation for a code sample showing the use of LOB reads and writes.
> - OCI Demonstration Programs
> - Runtime Data Allocation and Piecewise Operations in OCI for general information about piecewise OCI operations
> - Polling Mode Operations in OCI

## E.3.9 OCILobTrim()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

**Purpose**

Truncates the LOB value to a shorter length. This function is deprecated. Use OCILobTrim2().

**Syntax**

```
sword OCILobTrim ( OCISvcCtx       *svchp,
                   OCIError        *errhp,
                   OCILobLocator   *locp,
                   ub4              newlen );
```

**Parameters**

**Related Topics**

- OCILobTrim2()
  Truncates the LOB value to a shorter length. This function must be used for LOBs of size greater than 4 GB.

## E.3.10 OCILobWrite()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

### Purpose

Writes a buffer into a LOB. This function is deprecated. Use `OCILobWrite2()`.

### Syntax

```
sword OCILobWrite ( OCISvcCtx       *svchp,
                    OCIError        *errhp,
                    OCILobLocator   *locp,
                    ub4             *amtp,
                    ub4              offset,
                    void            *bufp,
                    ub4              buflen,
                    ub1              piece,
                    void            *ctxp,
                    OCICallbackLobWrite      (cbfp)
                                    (
                                      void     *ctxp,
                                      void     *bufp,
                                      ub4      *lenp,
                                      ub1      *piecep
                                    )
                    ub2              csid,
                    ub1              csfrm );
```

### Parameters

**svchp (IN/OUT)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to OCIErrorGet() for diagnostic information when there is an error.

**locp (IN/OUT)**
An internal LOB locator that uniquely references the LOB. This locator must have been 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 Table E-6.

**Table E-6 Characters or Bytes in amtp for OCILobWrite()**

| LOB or BFILE | Input with Fixed-Width Client-Side Character Set | Input with Varying-Width Client-Side Character Set | Output |
|---|---|---|---|
| `BLOB`s and `BFILE`s | bytes | bytes | bytes |
| `CLOB`s and `NCLOB`s | 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`. If 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, the parameter `amtp` contains 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), specify the offset in the first call; in subsequent polling calls, the offset parameter is ignored. When you use a callback, there is no offset parameter.

**bufp (IN)**
The pointer to a buffer from which the piece is 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 results.

**buflen (IN)**
The length, in bytes, of the data in the buffer. This value differs from the `amtp` value for `CLOB`s and `NCLOB`s when the `amtp` parameter is specified in terms of characters, and 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 that the buffer is 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 can be registered to be called for each piece in a piecewise write. If this is `NULL`, the standard polling method is 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 the 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.
The `csfrm` parameter 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 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 you read or write LOBs, specify a character set form (`csfrm`) that matches the form of the locator itself.

When you use the polling mode for `OCILobWrite()`, the first call must specify values for `offset` and `amtp`, but on subsequent polling calls to `OCILobWrite()`, you 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 is invoked to get the next piece after a piece is written to the pipe. Each piece is 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 Database (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. `BFILE`s are not allowed, because 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 `CLOB`s and `NCLOB`s. 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`. If 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 UTF-16 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 before 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 before performing this operation, then you must close it before you commit or roll back 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

**Related Topics**

- OCILobWrite2()
  Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB.

**See Also:**

- PL/SQL REF CURSORs and Nested Tables in OCI for additional information about Unicode format
- The demonstration programs included with your Oracle Database installation for a code sample showing the use of LOB reads and writes.
- OCI Demonstration Programs
- Runtime Data Allocation and Piecewise Operations in OCI for general information about piecewise OCI operations
- Polling Mode Operations in OCI

## E.3.11 OCILobWriteAppend()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

### Purpose

Writes data starting at the end of a LOB. This function is deprecated. Use OCILobWriteAppend2().

### Syntax

```
sword OCILobWriteAppend ( OCISvcCtx     *svchp,
                          OCIError      *errhp,
                          OCILobLocator *locp,
                          ub4           *amtp,
                          void          *bufp,
                          ub4            buflen,
                          ub1            piece,
                          void          *ctxp,
                          OCICallbackLobWrite       (cbfp)
                                         (
                                           void     *ctxp,
                                           void     *bufp,
                                           ub4      *lenp,
                                           ub1      *piecep
                                         )
                          ub2            csid,
                          ub1            csfrm );
```

### Parameters

**svchp (IN)**
The service context handle.

**errhp (IN/OUT)**
An error handle that you can pass to OCIErrorGet() for diagnostic information when there is 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 Table E-7.

**Table E-7 Characters or Bytes in amtp for OCILobWriteAppend()**

| LOB or BFILE | Input with Fixed-Width Client-Side Character Set | Input with Varying-Width Client-Side Character Set | Output |
|---|---|---|---|
| `BLOB`s and `BFILE`s | bytes | bytes | bytes |
| `CLOB`s and `NCLOB`s | 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`. If 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 is specified on input, and the data is written in pieces, the parameter `amtp` contains 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 `CLOB`s or `NCLOB`s.

**bufp (IN)**
The pointer to a buffer from which the piece is 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 results.

**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 that the buffer is 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 can be registered to be called for each piece in a piecewise write. If this is `NULL`, the standard polling method is 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 the 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.
The `csfrm` parameter 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 is invoked to get the next piece after a piece is written to the pipe. Each piece is 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 before 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 before performing this operation, then you must close it before you commit or roll back 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, Oracle recommends that you enclose write operations to the LOB within the open or close statements.

**Related Topics**

- OCILobWriteAppend2()
  Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.

> **See Also:**
>
> About Improving LOB Read/Write Performance

## E.4 Deprecated Database Advanced Queuing Functions

Describes deprecated Oracle Database Advanced Queuing functions.

Table E-8 lists the deprecated Database Advanced Queuing functions that are described in this section.

**Table E-8 Deprecated Database Advanced Queuing Functions**

| Function | Purpose |
|---|---|
| OCIAQListen() | Listen on one or more queues on behalf of a list of agents. |

### E.4.1 OCIAQListen()

This function was deprecated in a release previous to Oracle Database 11*g* Release 2 (11.2).

**Purpose**

Listens on one or more queues on behalf of a list of agents. This function is deprecated. Use`OCIAQListen2()`.

**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 that you can pass to OCIErrorGet() for diagnostic information when there is 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 Topics

- OCIAQListen2()
  Listens on one or more queues on behalf of a list of agents.

# F

# Multithreaded extproc Agent

What is the multithreaded extproc agent?

This appendix explains what the multithreaded `extproc` agent is, how it contributes to the overall efficiency of a distributed database system, and how to administer it.

This section includes the following topics:

- Why Use the Multithreaded extproc Agent?
- Multithreaded extproc Agent Architecture
- Administering the Multithreaded extproc Agent

## F.1 Why Use the Multithreaded extproc Agent?

This section explains how the multithreaded `extproc` agent contributes to the efficiency of external procedures.

This section includes the following topics:

- The Challenge of Dedicated Agent Architecture
- The Advantage of Multithreading

### F.1.1 The Challenge of Dedicated Agent Architecture

By default, an `extproc` agent is started for each user session and the `extproc` agent process terminates only when the user session ends.

This architecture can consume an unnecessarily large amount of system resources. For example, suppose that several thousand user sessions simultaneously spawn `extproc` agent processes. Because an `extproc` agent process is started for each session, several thousand `extproc` agent processes run concurrently. The `extproc` agent processes operate regardless of whether each individual `extproc` agent process is active at the moment. Thus `extproc` agent processes and open connections can consume a disproportionate amount of system resources. When sessions connect to Oracle Database, this problem is addressed by starting the server in shared server mode. Shared server mode allows database connections to be shared by a small number of server processes.

### F.1.2 The Advantage of Multithreading

The Oracle Database shared server architecture assumes that even when several thousand user sessions are open, only a small percentage of these connections are active at any given time.

In shared server mode, there is a pool of shared server processes. User sessions connect to dispatcher processes that place the requested tasks in a queue. The tasks are picked up by the first available shared server processes. The number of shared server processes is usually less that the number of user sessions.

The multithreaded `extproc` agent provides similar functionality for connections to external procedures. The multithreaded `extproc` agent architecture uses a pool of shared agent threads. The tasks requested by the user sessions are put in a queue and are picked up by the first available multithreaded `extproc` agent thread. Because only a small percentage of user connections are active at a given moment, using a multithreaded `extproc` architecture allows more efficient use of system resources.

## F.2 Multithreaded extproc Agent Architecture

One multithreaded `extproc` agent must be started for each system identifier (SID) before attempting to connect to the external procedure. This is done using the agent control utility `agtctl`. This utility is also used to configure the agent and to shut down the agent.

Each Oracle Net listener that is running on a system listens for incoming connection requests for a set of SIDs. If the SID in an incoming Oracle Net connect string is an SID for which the listener is listening, then that listener processes the connection. Further, if a multithreaded `extproc` agent was started for the SID, then the listener passes the request to that `extproc` agent.

In the architecture for multithreaded `extproc` agents, each incoming connection request is processed by different kinds of threads:

- A single **monitor** thread. The monitor thread is responsible for:
  - Maintaining communication with the listener
  - Monitoring the load on the process
  - Starting and stopping threads when required
- Several **dispatcher** threads. The dispatcher threads are responsible for:
  - Handling communication with the Oracle Database
  - Passing task requests to the task threads
- Several **task** threads. The task threads handle requests from the Oracle Database processes.

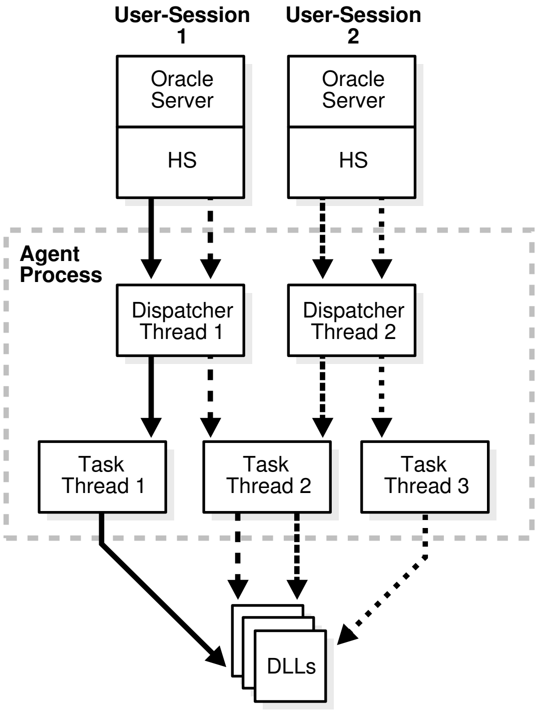
Figure F-1 illustrates the architecture of the multithreaded `extproc` agent. User sessions 1 and 2 issue requests for callouts to functions in some DLLs. These requests get serviced through heterogeneous services to the multithreaded `extproc` agent. These requests get handled by the agent's dispatcher threads, which then pass them on to the task threads. The task thread that is actually handling a request is responsible for loading the respective DLL and calling the function therein.

- All requests from a user session get handled by the same dispatcher thread. For example, dispatcher 1 handles communication with user session 1, and dispatcher 2 handles communication with user session 2. This is the case for the lifetime of the session.
- The individual requests can be serviced by different task threads. For example, task thread 1 can handle the request from user session 1, and later handle the request from user session 2.

**See Also:**

*Oracle Database Administrator’s Guide*. for details on managing processes for external procedures

**Figure F-1 Multithreaded extproc Agent Architecture**


User-Session 1
User-Session 2
Oracle Server
HS
Oracle Server
HS
Agent Process
Dispatcher Thread 1
Dispatcher Thread 2
Task Thread 1
Task Thread 2
Task Thread 3
DLLs

These three thread types roughly correspond to the Oracle Database multithreaded server PMON, dispatcher, and shared server processes, respectively.

**Note:**

All requests from a user session go through the same dispatcher thread, but can be serviced by different task threads. Also, several task threads can use the same connection to the external procedure.

These topics explain each type of thread in more detail:

- Monitor Thread
- Dispatcher Threads
- Task Threads

> **See Also:**
>
> Administering the Multithreaded extproc Agent for more information about starting and stopping the multithreaded `exproc` agent by using the agent control utility `agtctl`

## F.2.1 Monitor Thread

When the agent control utility `agtctl` starts a multithreaded `extproc` agent for a SID, `agtctl` creates the monitor thread.

The monitor thread performs these functions:

- Creates the dispatcher and task threads.
- Registers the dispatcher threads with all the listeners that are handling connections to this `extproc` agent. While the dispatcher for this SID is running, the listener does not start a process when it gets an incoming connection. Instead, the listener gives the connection to this same dispatcher.
- Monitors the other threads and sends load information about the dispatcher threads to all the listener processes handling connections to this `extproc` agent, enabling listeners to give incoming connections to the least loaded dispatcher.
- Continues to monitor each of the threads it has created.

## F.2.2 Dispatcher Threads

How are dispatcher threads used?

Dispatcher threads perform these functions:

- Accept incoming connections and task requests from Oracle Database servers.
- Place incoming requests on a queue for a task thread to pick up.
- Send results of a request back to the server that issued the request.

> **Note:**
>
> After a user session establishes a connection with a dispatcher, all requests from that user session go to the same dispatcher until the end of the user session.

## F.2.3 Task Threads

How are task threads used?

Task threads perform these functions:

- Pick up requests from a queue.
- Perform the necessary operations.
- Place the results on a queue for a dispatcher to pick up.

# F.3 Administering the Multithreaded extproc Agent

One multithreaded `extproc` agent must be started for each system identifier (SID) before attempting to connect to the external procedure.

A multithreaded `extproc` agent is started, stopped, and configured by an agent control utility called `agtctl`, which works like `lsnrctl`. However, unlike `lsnrctl`, which reads a configuration file (`listener.ora`), `agtctl` takes configuration information from the command line and writes it to a control file.

Before starting `agtctl`, ensure that Oracle Listener is running. Then use the `agtctl` commands to set the `agtctl` configuration parameters (if you do not want their default values) and to start `agtctl`, as in Example F-1.

You can use `agtctl` commands in either single-line command mode or shell mode.

**Topics:**

- Agent Control Utility (agtctl) Commands
- About Using agtctl in Single-Line Command Mode
- Using Shell Mode Commands
- Configuration Parameters for Multithreaded extproc Agent Control

**Example F-1 Setting Configuration Parameters and Starting agtctl**

```
agtctl set max_dispatchers  2 ep_agt1
agtctl set tcp_dispatchers  1   ep_agt1
agtctl set max_task_threads 2  ep_agt1
agtctl set max_sessions     5 ep_agt1
agtctl unset listener_address ep_agt1
agtctl set listener_address "(address=(protocol=ipc)(key=extproc))" ep_agt1
agtctl startup extproc ep_agt1
```

## F.3.1 Agent Control Utility (agtctl) Commands

What are the Agent Control Utility commands and what are they used for?

You can start and stop `agtctl` and create and maintain its control file by using the commands shown in Table F-1.

**Table F-1 Agent Control Utility (agtctl) Commands**

| Command | Description |
|---|---|
| `startup` | Starts a multithreaded `extproc` agent |
| `shutdown` | Stops a multithreaded `extproc` agent |
| `set` | Sets a configuration parameter for a multithreaded `extproc` agent |
| `unset` | Causes a parameter to revert to its default value |
| `show` | Displays the value of a configuration parameter |
| `delete` | Deletes the entry for a particular SID from the control file |
| `exit` | Exits shell mode |

**Table F-1 (Cont.) Agent Control Utility (agtctl) Commands**

| Command | Description |
|---|---|
| `help` | Lists available commands |

These commands can be issued in one of two ways:

- You can issue commands from the UNIX or DOS shell. This mode is called single-line command mode.
- You can enter `agtctl` and an `AGTCTL>` prompt appears. You then can enter commands from within the `agtctl` shell. This mode is called shell mode.

The syntax and parameters for `agtctl` commands depend on the mode in which they are issued.

> **Note:**
>
> - All commands are case-sensitive.
> - The `agtctl` utility puts its control file in the directory specified by either one of two environment variables, `AGTCTL_ADMIN` or `TNS_ADMIN`. Ensure that at least one of these environment variables is set and that it specifies a directory to which the agent has access.
> - If the multithreaded `extproc` agent requires that an environment variable be set, or if the `ENVS` parameter was used when configuring the `listener.ora` entry for the agent working in dedicated mode, then all required environment variables must be set in the UNIX or DOS shell that runs the `agtctl` utility.

## F.3.2 About Using agtctl in Single-Line Command Mode

This section describes the use of `agtctl` commands.

They are presented in single-line command mode.

This section includes the following topics:

- Setting Configuration Parameters for a Multithreaded extproc Agent
- Starting a Multithreaded extproc Agent
- Shutting Down a Multithreaded extproc Agent
- Examining the Value of Configuration Parameters
- Resetting a Configuration Parameter to Its Default Value
- Deleting an Entry for a Specific SID from the Control File
- Requesting Help

### F.3.2.1 Setting Configuration Parameters for a Multithreaded extproc Agent

Set the configuration parameters for a multithreaded `extproc` agent before you start the agent.

If a configuration parameter is not specifically set, a default value is used. Configuration parameters and their default values are shown in Table F-2.

Use the `set` command to set multithreaded `extproc` agent configuration parameters.

**Syntax**

```
agtctl set parameter parameter_value agent_sid
```

*`parameter`* is the parameter that you are setting.

*`parameter_value`* is the value being assigned to that parameter.

*`agent_sid`* is the SID that this agent services. This must be specified for single-line command mode.

**Example**

```
agtctl set max_dispatchers 5 salesDB
```

### F.3.2.2 Starting a Multithreaded extproc Agent

Use the `startup` command to start a multithreaded `extproc` agent.

**Syntax**

```
agtctl startup extproc agent_sid
```

*`agent_sid`* is the SID that this multithreaded `extproc` agent services. This must be specified for single-line command mode.

**Example**

```
agtctl startup extproc salesDB
```

### F.3.2.3 Shutting Down a Multithreaded extproc Agent

Use the `shutdown` command to stop a multithreaded `extproc` agent.

There are three forms of shutdown:

- Normal (default)

  `agtctl` asks the multithreaded `extproc` agent to terminate itself gracefully. All sessions complete their current operations and then shut down.

- Immediate

  `agtctl` tells the multithreaded `extproc` agent to terminate immediately. The agent exits immediately regardless of the state of current sessions.

- Abort

Without talking to the multithreaded `extproc` agent, `agtctl` issues a system call to stop it.

**Syntax**

```
agtctl shutdown [immediate|abort] agent_sid
```

*`agent_sid`* is the SID that the multithreaded `extproc` agent services. It must be specified for single-line command mode.

**Example**

```
agtctl shutdown immediate salesDB
```

### F.3.2.4 Examining the Value of Configuration Parameters

To examine the value of a configuration parameter, use the `show` command.

**Syntax**

```
agtctl show parameter agent_sid
```

*`parameter`* is the parameter that you are examining.

*`agent_sid`* is the SID that this multithreaded `extproc` agent services. This must be specified for single-line command mode.

**Example**

```
agtctl show max_dispatchers salesDB
```

### F.3.2.5 Resetting a Configuration Parameter to Its Default Value

You can reset a configuration parameter to its default value using the `unset` command.

**Syntax**

```
agtctl unset parameter agent_sid
```

*`parameter`* is the parameter that you are resetting (or changing).

*`agent_sid`* is the SID that this multithreaded `extproc` agent services. It must be specified for single-line command mode.

**Example**

```
agtctl unset max_dispatchers salesDB
```

### F.3.2.6 Deleting an Entry for a Specific SID from the Control File

The `delete` command deletes the entry for the specified SID from the control file.

**Syntax**

```
agtctl delete agent_sid
```

*`agent_sid`* is the SID entry to delete.

**Example**

```
agtctl delete salesDB
```

### F.3.2.7 Requesting Help

Use the `help` command to view a list of available commands for `agtctl` or to see the syntax for a particular command.

**Syntax**

```
agtctl help [command]
```

*`command`* is the name of the command whose syntax you want to view. The default is all `agtctl` commands.

**Example**

```
agtctl help set
```

## F.3.3 Using Shell Mode Commands

When issuing the `AGTCTL>` command, it results in the prompt `AGTCTL>`.

In shell mode, start `agtctl` by entering:

```
agtctl
```

Thereafter, because you are issuing commands from within the `agtctl` shell, you need not prefix the command string with `agtctl`.

Set the name of the agent SID by entering:

```
AGTCTL> set agent_sid agent_sid
```

All subsequent commands are assumed to be for the specified SID until the `agent_sid` value is changed. Unlike single-line command mode, you do not specify *`agent_sid`* in the command string.

You can set the language for error messages as follows:

```
AGTCTL> set language language
```

The commands themselves are the same as those for the single-line command mode. To exit shell mode, enter `exit`.

The following examples use shell mode commands:

- Example: Setting a Configuration Parameter
- Example: Starting a Multithreaded extproc Agent

### F.3.3.1 Example: Setting a Configuration Parameter

How to set the value for the `shutdown_address` configuration parameter.

This example sets a value for the `shutdown_address` configuration parameter.

```
AGTCTL> set shutdown_address (address=(protocol=ipc)(key=oraDBsalesDB))
```

### F.3.3.2 Example: Starting a Multithreaded extproc Agent

How to start a multithreaded `extproc` agent.

This example starts a multithreaded `extproc` agent.

```
AGTCTL> startup extproc
```

## F.3.4 Configuration Parameters for Multithreaded extproc Agent Control

What do the configuration parameters for multithreaded extproc agent control do?

Table F-2 describes and gives the defaults of the configuration parameters for the agent control utility.

**Table F-2    Configuration Parameters for agtctl**

| Parameter | Description | Default Value |
|---|---|---|
| `max_dispatchers` | Maximum number of dispatchers | 1 |
| `tcp_dispatchers` | Number of dispatchers listening on TCP (the rest are using IPC) | 0 |
| `max_task_threads` | Maximum number of task threads | 2 |
| `max_sessions` | Maximum number of sessions for each task thread | 5 |
| `listener_address` | Address on which the listener is listening (needed for registration) | |

`listener_address`

```
(ADDRESS_LIST=
    (ADDRESS=
        (PROTOCOL=IPC)
        (KEY=PNPKEY))
    (ADDRESS=
        (PROTOCOL=IPC)
        (KEY=listener_sid))
    (ADDRESS=
        (PROTOCOL=TCP)
        (HOST=127.0.0.1)
        (PORT=1521)))
```

Note: *`listener_sid`* is the IPC key of the address, on the Oracle Database, on which the listener is listening.

**Table F-2 (Cont.) Configuration Parameters for agtctl**

| Parameter | Description | Default Value |
|---|---|---|
| `shutdown_address` | Address the agent uses to communicate with the listener. This is the address on which the agent listens for all communication, including shutdown messages from `agtctl`. | Notes:<br>- *`agent_sid`* is the SID of the multithreaded `extproc` agent.<br>- \|\| indicates that *`listener_sid`* and *`agent_sid`* are concatenated into one string. |

`shutdown_address`

```
(ADDRESS=
    (PROTOCOL=IPC)
    (KEY=listener_sid || agent_sid))
(ADDRESS=
    (PROTOCOL=TCP)
    (HOST=127.0.0.1)
    (PORT=1521))
```

### max_dispatchers, tcp_dispatchers, max_task_threads, and max_sessions

To improve performance, you might need to change the values of some or all of the parameters `max_dispatchers`, `tcp_dispatchers`, `max_task_threads`, and `max_sessions`.

You can calculate the optimum values of `max_dispatchers`, `tcp_dispatchers`, `max_task_threads` with these formulas:

```
max_dispatchers = CEIL(x/y)
tcp_dispatchers = CEIL(x_tcpip/y)
max_task_threads = CEIL(x/max_sessions)
```

Where:

- `CEIL` is a SQL function that returns the smallest integer greater than or equal to its argument.
- *x* is the maximum number of sessions that can be connected to `extproc` concurrently.
- *y* is the maximum number of connections that the system can support for each dispatcher.
- *`x_tcpip`* is the maximum number of sessions that can be connected to `extproc` concurrently by TCP/IP.

  (*x* - *`x_tcpip`* is the maximum number of sessions that can be connected to `extproc` concurrently by IPC.)

There is no formula for computing the optimum value of `max_sessions`, which affects `max_task_threads`.

You must fine-tune these parameter settings, based on the capability of your hardware, and ensure that the concurrent threads do not exhaust your operating system.

The value of `max_dispatchers` must be at least 1 (which is the default).

### Example

Suppose:

- The maximum number of sessions that can be connected to `extproc` concurrently (*x*) is 650.
- The maximum number of sessions that can be connected to `extproc` concurrently by TCP/IP (*x_tcpip*) is 400.

  (The maximum number of sessions that can be connected to `extproc` concurrently by IPC is 650-400=250.)
- The maximum number of connections that the system can support for each dispatcher (*y*) is 100.
- The maximum number of sessions for each task thread (`max_sessions`) is 20.

The optimum values for these parameters are:

```
max_dispatchers = CEIL(650/100) = CEIL(6.5)  = 7
tcp_dispatchers  = CEIL(400/100) = CEIL(4)    = 4
max_task_threads = CEIL(650/20)  = CEIL(32.5) = 33
```

That is, optimally:

- The maximum number of dispatchers is seven.
- Four of the seven dispatchers are listening on TCP/IP, and the remaining three are listening on IPC.
- The maximum number of task threads is 33.

### listener_address and shutdown_address

The values of the configuration parameters `listener_address` and `shutdown_address` are specified with `ADDRESS`, as shown in both Table F-2 and Example F-1. Within `ADDRESS`, you can specify the parameter `HOST`, which can be either an IPv6 or IPv4 address or a host name. If `HOST` is a host name, then these values of the optional `ADDRESS` parameter `IP` are relevant:

| IP Value | Meaning |
|---|---|
| `FIRST` | Listen on the first IP address returned by the DNS resolution of the host name. |
| `V4_ONLY` | Listen only on the IPv4 interfaces in the system. |
| `V6_ONLY` | Listen only on the IPv6 interfaces in the system. |

For example, this value of `listener_address` or `shutdown_address` restricts it to IPv6 interfaces:

```
"(ADDRESS=(PROTOCOL=tcp)(HOST=sales)(PORT=1521)(IP=V6_ONLY))"
```

**See Also:**

*Oracle Database Net Services Administrator's Guide* for detailed information about IPv6 support in Oracle Database

# Index

## Numerics

64-bit integer support, *5-13*

## A

administration handle, *16-1*
    attributes, *A-34*
    description, *16-1*
ADR, *3-25*
ADR base location, *3-25*
ADR, controlling creation, *3-29*
ADRCI command-line tool, *3-27*
ADT
    *See* object type
advanced queuing
    agent descriptor, *4-12*
    dequeue options descriptor, *4-12*
    descriptor, *4-16*
    enqueue options descriptor, *4-12*
    listen options descriptor, *4-12*
    message properties descriptor, *4-12*
    notification descriptor, *4-12*
advantages
    OCI, *1-1*
Agent Control Utility (agtctl)
    commands
        in shell mode, *F-9*
        in single-line mode, *F-6*
        list of, *F-5*
    extproc administration and, *F-5*
    extproc architecture and, *F-2*
allocation duration
    example, *22-16*
    of objects, *22-15*
ALTER SESSION SET CONTAINER, *17-2*, *17-3*
    enhancements on OCI calls with, *17-2*
    restrictions on OCI calls with, *17-3*
ALTER SESSION SWITCH CONTAINER SWITCH SERVICE
    restrictions on OCI calls with, *17-5*
alternatives to OCI
    OCCI, *1-3*
    ODPI-C, *1-3*
    Pro*C/C++, *1-3*
ANSI DATE descriptor, *4-12*
AnyData, *20-26*
AnyDataSet, *20-26*
AnyType, *20-26*
application context, *10-28*
Application Continuity, *12-22*
    constructs not supported, *12-27*
    failed replay, *12-25*
    following a recoverable error, *12-23*
    successful replay criteria, *12-24*
    successful replay side effects to be aware of, *12-27*
    supported OCI functions for fail over, *12-26*
    what disables it in OCI, *12-24*
    when it is most effective, *12-25*
application initialization, *4-19*
AQ
    *See* Oracle Database Advanced Queuing.
argument attributes, *8-20*
arrays
    binds, *20-35*
    defines, *20-39*
    DML, maximum rows, *7-4*
    skip parameter for, *7-24*
arrays of structures, *7-22*
    indicator variables, *7-25*
    OCI calls used, *7-24*
    skip parameters, *7-23*
atomic NULLs, *19-28*
attribute descriptor object, *20-25*
attributes
    administration handle, *A-34*
    authentication handle, *A-20*
    authentication information handle, *A-21*
    bind handle, *A-51*
    complex object retrieval (COR) descriptor, *A-74*
    complex object retrieval (COR) handle, *A-74*
    connection pool handle, *A-34*
    continuous query notification, *A-92*
    continuous query notification descriptor, *A-93*
    define handle, *A-54*
        description, *A-54*
    describe handle, *A-57*

attributes *(continued)*
    direct path loading
        column array handle, *A-108*
        column parameter, *A-110*
        context handle, *A-99*
        function context handle, *A-106*
        handle, *A-98*
        stream handle, *A-109*
    environment handle, *A-3*
    error handle, *A-10*
    event handle, *A-116*
    LOB locator, *A-73*
    notification descriptor, *A-96*
    OCIAQAgent descriptor, *A-85*
    OCIAQDeqOptions descriptor, *A-77*
    OCIAQEnqOptions descriptor, *A-75*
    OCIAQMsgProperties descriptor, *A-81*
    OCIServerDNs descriptor, *A-85*
    of handles, *4-11*
    of objects, *19-17*
    of parameter descriptors, *8-5*
    of parameters, *8-5*
    parameter descriptor, *A-58*
    process handle, *A-114*
    server handle, *A-15*
    service context handle, *A-10*
    session pool handle, *A-36*
    Shard Instance Descriptor, *A-59*
    statement handle, *A-41*
    subscription handle, *A-86*
    transaction handle, *A-40*
    user session handle, *A-21*
authentication
    by Distinguished Name, *10-17*
    by X.509 Certificate, *10-17*
    management, *10-11*
authentication handle
    attributes, *A-20*
authentication information handle, *4-3*
authorize functions, *25-3*
auto_tune, *15-14*
auto-tuning
    client session features
        comparison of auto-tuning parameters, *15-16*
        enabling and disabling, *15-19*
        usage examples of client attributes, *15-18*
    OCI client attributes, *15-13*
    OCI client session features
        benefits of, *15-13*
    OCI client statement cache, *15-12*
Automatic Diagnostic Repository (ADR), *3-25*

## B

BASICFILE parameter, *9-28*
batch error mode, *6-9*
batch jobs, authenticating users in, *10-13*
BFILE
    data type, *5-25*
BFILE data type locator descriptor, *4-12*
BINARY_DOUBLE data type, *5-7*
BINARY_FLOAT data type, *5-7*
bind and define in statement caching, *15-5*
bind functions, *25-59*
bind handle
    attributes, *A-51*
    description, *4-8*
bind operations, *6-6*, *7-1*, *20-33*
    associations made, *7-1*
    example, *7-6*
    initializing variables, *7-1*
    LOBs, *7-10*
    named data types, *20-33*
    named versus positional, *7-3*
    OCI array interface, *7-4*
    OCI_DATA_AT_EXEC mode, *7-16*
    PL/SQL, *7-5*
    positional versus named, *7-3*
    REF CURSOR variables, *7-17*
    REFs, *20-34*
    steps used, *7-6*
bind placeholders, *6-7*
binding
    arrays, *20-35*
    buffer expansion, *7-38*
    multiple buffers, *7-25*
    OCINumber, *20-41*
    PL/SQL placeholders, *4-38*
    summary, *7-9*
BLOB data type, *5-26*
blocking modes, *4-36*
branches
    detaching, *10-6*
    preparing multiple, *10-8*
    resuming, *10-6*
buffer expansion during binding, *7-38*
buffered messaging, *13-30*
buffering LOB operations, *9-12*
building OCI applications, *B-1*

## C

C data types
    manipulating with OCI, *20-4*
cache functions
    server round-trips, *C-5*
cache, client result, *15-12*

callbacks
    dynamic registrations, *14-6*
    for LOB operations, *9-15*
    for reading LOBs, *9-15*
    for writing LOBs, *9-17*
    from external procedures, *14-11*
    LOB streaming interface, *9-15*
    parameter modes, *25-98*
    registration for transparent application failover, *12-11*
    restrictions, *14-8*
    user-defined functions, *14-1*
canceling a cursor, *26-5*
canceling OCI calls, *4-33*
cartridge functions, *29-1*
CASE OTT parameter, *24-29*
CDBs
    enhancements on OCI API calls with, *17-1*
    restrictions on OCI API calls with, *17-2*
change notification descriptor, *4-12*
CHAR
    external data type, *5-20*
character set conversion of Unicode, *4-45*
character set form, *7-34*
character set ID, *7-34*
    Unicode, *A-51*, *A-54*
character sets
    OCI SODA client, *36-1*
    XStream OCI interface, *34-2*
character-length semantics, *7-38*–*7-40*, *8-26*
CHARZ
    external data type, *5-21*
checkerr() listing, *26-155*
client and server interoperability, *3-20*
client auto-tuning
    comparison of auto-tuning parameters, *15-16*
    OCI client attributes, *15-13*
    session features
        benefits of, *15-13*
        enabling and disabling, *15-19*
    usage examples of client attributes, *15-18*
client result cache, *15-12*
client statement cache auto-tuning
    OCI client session feature, *15-12*
CLIENTCONTEXT namespace, *10-29*
CLOB
    data type, *5-26*
code
    example programs, *B-1*
    list of demonstration programs, *B-1*
CODE OTT parameter, *24-27*
coding guidelines
    reserved words, *4-35*
coherency
    of object cache, *22-4*
collections
    attributes, *8-13*
    data manipulation functions, *20-19*
    describing, *8-1*
    description, *20-17*
    functions for manipulating, *20-18*
    multilevel, *20-22*
    scanning functions, *20-19*
column objects
    direct path loading of, *21-18*
columns
    attributes, *8-4*, *8-16*
commit, *4-25*
    in object applications, *22-15*
    one-phase for global transactions, *10-7*
    two-phase for global transactions, *10-7*
compatibility
    of releases in OCI, *3-20*
compiling
    OCI application, *D-2*
    OCI with Oracle XA, *D-5*
    XA Library, *D-4*
complex object descriptor, *4-12*
complex object retrieval, *19-20*
    descriptor attributes, *A-74*
    handle, *4-9*
    handle attributes, *A-74*
    implementing, *19-23*
    navigational prefetching, *19-24*
complex object retrieval (COR) descriptor, *4-16*
CONFIG OTT parameter, *24-28*
connect functions, *25-3*
connection, *4-19*
connection mode
    nonblocking, *4-36*
connection pool handle
    attributes, *A-34*
    description, *4-11*
    initializing, *11-18*
connection pooling, *11-15*, *11-23*
    code example, *11-23*
    creating, *11-18*
    database resident, *11-11*
connection pools and TAF, *11-17*
consistency
    of object cache, *22-4*
continuous query notification, *13-1*
    attributes, *A-92*
    descriptor attributes, *A-93*
copying
    objects, *19-31*
COR
    *See* complex object retrieval
creating
    objects, *19-31*

cursor cancellation, *26-5*
custom pools, *18-7*

## D

data cartridges
  OCI functions, *4-1*, *29-1*
data definition language, *1-5*
  SQL statements, *1-5*
data manipulation language, *1-6*
  SQL statements, *1-6*
data structures, *4-3*
    *See also* descriptors
data types
  ANSI DATE, *5-27*
  BFILE, *5-25*
  binding and defining, *20-39*
  BLOBs (binary large objects), *5-26*
  CLOB, *5-26*
  conversions, *5-29*
  direct path loading, *21-3*, *A-110*
  external, *5-3*, *5-7*
  FILE, *5-25*
  for piecewise operations, *7-46*
  internal, *5-4*
  internal codes, *5-4*
  INTERVAL DAY TO SECOND, *5-28*
  INTERVAL YEAR TO MONTH, *5-28*
  manipulating with OCI, *20-4*
  mapping and manipulation functions, *C-7*
  mapping from Oracle to C, *20-2*
  mapping, Oracle methodology, *20-3*
  mapping, OTT utility, *24-9*
  NCLOB, *5-26*
  Oracle, *5-1*
  TIMESTAMP, *5-27*
  TIMESTAMP WITH LOCAL TIME ZONE, *5-28*
  TIMESTAMP WITH TIME ZONE, *5-27*
database connection
  for object applications, *19-10*
database resident connection pooling, *11-11*
database type attributes
  type OCI_PTYPE_DATABASE, *8-23*
databases
  attributes, *8-23*
  describing, *8-1*
  shutting down, *16-1*
  starting up, *16-1*
DATE
  external data type, *5-17*
date cache, *21-14*
date descriptor, *4-16*
DATE, ANSI
  data type, *5-27*
datetime
  avoiding unexpected results, *5-28*
datetime and date
  migration rules, *5-33*
datetime descriptor, *4-16*
DDL
  *See* data definition language
DEDUPLICATE, *9-28*
default file name extensions
  OTT utility, *24-36*
default name mapping
  OTT utility, *24-36*
define
  arrays, *20-39*
  return and error codes, *4-28*
define functions, *25-59*
define handle
  attributes, *A-54*
  description, *4-8*
define operations, *6-16*, *7-17*, *20-36*
  example, *7-18*
  LOBs, *7-20*
  named data types, *20-36*
  piecewise fetch, *7-22*
  PL/SQL output variables, *7-22*
  REFs, *20-37*
  steps used, *7-18*
defining
  multiple buffers, *7-25*
  OCINumber, *20-41*
deletes
  positioned, *4-34*
demonstration files location, *3-1*
demonstration programs, *B-1*, *D-2*
  list, *B-1*
describe
  explicit, *6-12*, *6-15*
  explicit and implicit, *8-4*
  implicit, *6-12*, *6-13*
  of collections, *8-1*
  of databases, *8-1*
  of packages, *8-1*
  of schemas, *8-1*
  of sequences, *8-1*
  of stored functions, *8-1*
  of stored procedures, *8-1*
  of synonyms, *8-1*
  of tables, *8-1*
  of types, *8-1*
  of views, *8-1*
  select list, *6-12*
describe functions, *25-59*
describe handle
  attributes, *A-57*
  description, *4-9*

describe operation
  server round-trips, *C-6*
describing the stored procedure
  example, *8-30*
descriptor, *4-12*
  advanced queuing
    agent, *4-12*
    dequeue options, *4-12*
    enqueue options, *4-12*
    listen options, *4-12*
    message properties, *4-12*
    notification, *4-12*
  allocating, *4-19*
  ANSI DATE, *4-12*
  BFILE data type locator, *4-12*
  change notification, *4-12*
  complex object, *4-12*
  complex object retrieval, *4-16*
  distinguished names of the database servers in a registration request, *4-12*
  functions, *25-46*
  INTERVAL DAY TO SECOND, *4-12*
  INTERVAL YEAR TO MONTH, *4-12*
  LOB data type locator, *4-12*
  objects, *20-25*
  parameter, *4-15*
  read-only parameter, *4-12*
  result set, *4-12*
  row change, *4-13*
  ROWID, *4-12*, *4-15*
  shard key and shard group key, *4-13*
  snapshot, *4-12*, *4-14*
  table change, *4-13*
  TIMESTAMP, *4-12*
  TIMESTAMP WITH LOCAL TIME ZONE, *4-12*
  TIMESTAMP WITH TIME ZONE, *4-12*
  user callback, *4-12*
detaching branches, *10-6*
DIAG_ADR_ENABLED sqlnet.ora parameter, *3-29*
DIAG_DDE_ENABLED sqlnet.ora parameter, *3-29*
DIAG_RESTRICTED, *3-29*
DIAG_SIGHANDLER_ENABLED, *3-29*
direct path loading, *21-1*
  attribute
    OCI_ATTR_DIRPATH_NO_INDEX_ERRORS, *21-37*
  column array handle attributes, *A-108*
  column parameter attributes, *A-110*
  context handle attributes, *A-99*
  data types of columns, *21-3*, *A-110*
  direct path column array handle, *21-6*
  direct path context handle, *21-4*
direct path loading *(continued)*
  direct path function context, *21-5*
  direct path stream handle, *21-6*
  example, *21-9*
  function context handle attributes, *A-106*
  functions, *21-7*, *26-97*
  handle attributes, *A-98*
  handles, *4-10*, *21-4*
  in pieces, *21-32*
  limitations, *21-8*
  of date columns, *21-14*
  stream handle attributes, *A-109*
directory structures, *D-2*
dispatcher thread, *F-2*
distinguished names of the database servers in a registration request descriptor, *4-12*
DML
  *See* data manipulation language
DRCP (database resident connection pooling), *11-11*
duration
  example, *22-16*
  of objects, *22-15*
dynamic registration
  Oracle XA Library, *D-5*
dynamically linked applications, *3-20*

## E

edition-based redefinition, *10-31*
editions, *10-31*
embedded objects
  fetching, *19-15*
embedded SQL, *1-8*
  mixing with OCI calls, *1-8*
empty LOB, inserting, *9-19*
enable, *15-15*
enhanced DML array
  feature, *6-9*
enhancements
  on OCI API calls with CDBs in general, *17-1*
  on OCI calls with ALTER SESSION SET CONTAINER, *17-2*
environment handle
  attributes, *A-3*
  description, *4-6*
environment handle attribute, *4-41*
error checking example, *26-155*
error codes
  define calls, *4-28*
  navigational functions, *27-4*
error handle
  attributes, *A-10*
  description, *4-6*

errors
    handling, *4-27*
    handling in object applications, *19-35*
ERRTYPE OTT parameter, *24-28*
evaluation context
    attributes, *8-24*
evaluation context attributes, *8-24*
event handle attributes, *A-116*
example
    demonstration programs, *B-1*
executing SQL statements, *6-7*
execution
    against multiple servers, *6-5*
    modes, *6-9*
    snapshots, *6-8*
explicit describe, *6-12*, *8-27*
extensions
    OTT utility default file name, *24-36*
external data types, *5-3*, *5-7*
    CHAR, *5-20*
    CHARZ, *5-21*
    conversions, *5-29*
    DATE, *5-17*
    FLOAT, *5-15*
    INTEGER, *5-15*
    LOB, *5-23*
    LONG, *5-17*
    LONG RAW, *5-19*
    LONG VARCHAR, *5-19*
    LONG VARRAW, *5-19*
    named data types, *5-22*
    NUMBER, *5-12*
    RAW, *5-18*
    REF, *5-22*
    ROWID, *5-23*
    SQLT_BLOB, *5-23*
    SQLT_CLOB, *5-23*
    SQLT_NCLOB, *5-23*
    SQLT_NTY, *5-22*
    SQLT_REF, *5-22*
    STRING, *5-15*
    UNSIGNED, *5-19*
    VARCHAR, *5-17*
    VARCHAR2, *5-10*
    VARNUM, *5-16*
    VARRAW, *5-19*
external procedure functions
    return codes, *29-1*
    with_context type, *29-1*
external procedures
    OCI callbacks, *14-11*
externally initialized context, *10-22*
extproc agent, *F-1*

## F

failover
    callback example, *12-11*
failover callbacks, *12-7*
    structure and parameters, *12-10*
fault diagnosability, *3-25*
    read or write client driver layer name, *A-21*
featureRelease, client library version, *26-153*
fetch
    piecewise, *7-45*, *7-52*
fetch operation, *6-16*
    LOB data, *6-17*
    setting prefetch count, *6-17*
FILE
    associating with operating system file, *9-3*
    data type, *5-25*
FLOAT
    external data type, *5-15*
flushing, *22-11*
    object changes, *19-15*
    objects, *22-11*
freeing
    objects, *19-31*, *22-9*
functions
    attributes, *8-8*

## G

generic documentation references
    compiling and linking OCI application, *D-2*, *D-3*
    demonstration programs, *D-2*
    invoking OTT from the command line, *D-6*
    XA linking file names, *D-4*
global transactions, *10-3*
globalization support, *4-39*
    byte order is native to machine on which the application is running, *4-43*
    OCI functions, *4-1*
GTRID
    *See* transaction identifier

## H

HA event notification, *12-2*
handle attributes, *4-11*
    SODA collection, *A-62*
    SODA document, *A-60*
handle functions, *25-46*
handle type constants, *4-3*
handles, *4-3*
    administration handle, *16-1*, *A-34*
    advantages of, *4-5*
    allocating, *4-5*, *4-19*

handles *(continued)*
    bind handle, *4-8*, *A-51*
    C data types, *4-3*
    child freed when parent freed, *4-5*
    complex object retrieval handle, *4-9*, *A-74*
    connection pool handle, *4-11*, *11-18*, *A-34*
    define handle, *4-8*, *A-54*
    describe handle, *4-9*, *A-57*
    direct path loading, *4-10*
    environment handle, *4-6*, *A-3*
    error handle, *4-6*, *A-10*
    freeing, *4-5*
    process attributes, *A-114*
    server handle, *4-6*, *A-15*
    service context handle, *4-6*, *A-10*
    statement handle, *4-8*, *A-41*
    subscription, *4-10*, *13-4*
    transaction handle, *4-6*, *A-40*
    types, *4-3*
    user session handle, *4-6*, *A-21*
header files
    location of, *1-16*, *3-1*, *D-2*
    oratypes.h, *5-38*
HFILE OTT parameter, *24-28*

## I

implicit describe, *8-27*
implicit results
    OCI support for, *15-9*
inbound servers
    OCI interface, *34-2*
indicator variables, *4-31*
    arrays of structures, *7-25*
    for named data types, *4-31*, *4-32*
    for REFs, *4-31*, *4-32*
    named data type defines, *20-37*
    PL/SQL OUT binds, *20-37*
    REF defines, *20-37*
    with named data type bind, *20-35*
    with REF bind, *20-35*
init.ora security parameters, *10-32*
INITFILE OTT parameter, *24-27*
INITFUNC OTT parameter, *24-28*
initialize
    all buffers, *7-5*
    functions, *25-3*
inserting an empty LOB, *9-19*
inserts
    piecewise, *7-45*, *7-47*
Instant Client, *2-1*
Instant Client Light (English), *2-7*
INTEGER external data type, *5-15*
interacting with Oracle Database, *D-4*
internal codes for data types, *5-4*
internal data types, *5-4*
    conversions, *5-29*
INTERVAL DAY TO SECOND data type, *5-28*
INTERVAL DAY TO SECOND descriptor, *4-12*
interval descriptor, *4-16*
INTERVAL YEAR TO MONTH data type, *5-28*
INTERVAL YEAR TO MONTH descriptor, *4-12*
intype file
    providing when running OTT utility, *24-8*
    structure of, *24-31*
INTYPE File Assistant, *D-6*
INTYPE OTT parameter, *24-26*
IP address
    IPv4, *A-86*
    IPv6, *A-86*
IPv6 addressing, *13-4*, *A-86*

## K

KEEP_DUPLICATES, *9-28*
key words, *4-35*

## L

LCR column flags, *35-19*, *35-21*, *35-26*, *35-51*, *35-66*
LDAP registration of publish-subscribe notification, *13-9*
libraries
    oci.lib, *D-3*
linking
    OCI application, *D-3*
    OCI with Oracle XA, *D-5*
    XA Library, *D-4*
list attributes
    type OCI_PTYPE_LIST, *8-22*
lists
    attributes, *8-22*
lmsgen utility, *4-47*
LoadLibrary, *D-3*
LOB and LONG bind restrictions, *7-12*
LOB data type locator descriptor, *4-12*
LOB functions, *26-14*
    server round-trips, *C-3*
LOB locator, *4-14*
    attributes, *A-73*
LOBs
    amount and offset parameters, *26-14*
    attributes of transient objects, *9-4*
    binding, *7-10*
    buffering, *9-12*
    callbacks, *9-15*
    character sets, *26-14*
    creating, *9-2*
    creating temporary, *9-19*

LOBs *(continued)*
- defining, *7-20*
- duration of temporary, *9-20*
- example of temporary, *9-22*
- external data types, *5-23*
- failover does not work, *12-13*
- fetching data, *6-17*
- fixed-width character sets, *26-14*
- freeing temporary, *9-19*
- greater than 4 GB, *9-5*
- locator, *4-14*
- modifying, *9-2*
- OCI functions, *9-9*
- prefetching, *9-25*
- size maximum, *9-5*
- temporary, *9-18*
- varying-width character sets, *26-14*

locator, *4-12*
- for LOB data type, *4-14*

locking, *22-13*
- objects, *22-13*
- optimistic model, *22-14*

logical transaction ID (LTXID)
- Transaction Guard, *12-14*

LONG
- external data type, *5-17*

LONG RAW
- external data type, *5-19*

LONG VARCHAR
- external data type, *5-19*

LONG VARRAW
- external data type, *5-19*

## M

makefiles, *3-1*, *B-1*

marking objects, *22-10*

memory_target, *15-16*

meta-attributes
- of objects, *19-17*
- of persistent objects, *19-17*
- of transient objects, *19-20*

method descriptor object, *20-25*

method results
- type OCI_PTYPE_TYPE_RESULT, *8-20*

migration
- session, *10-11*, *25-28*

miscellaneous functions, *26-152*

mode
- agtctl command, *F-5*

monitor thread, *F-2*

multi-property tags, *11-3*
- PL/SQL fix-up callback example, *11-4*

multiple servers
- executing statement against, *6-5*

multithreaded development
- basic concepts, *10-34*

multithreaded extproc agent, *F-1*

multithreading, *10-34*

mutexes, *10-34*

## N

name-value pair attributes
- type OCI_PTYPE_NAME_VALUE, *8-26*

named data types
- binding, *20-33*
- binding and defining, *20-39*
- defining, *20-36*
- definition, *5-22*
- external data types, *5-22*
- indicator variables, *4-32*
- indicator variables for, *4-31*

native double, *5-29*

native float, *5-29*

navigation, *22-18*

navigational functions
- error codes, *27-4*
- return values, *27-4*
- terminology, *27-3*

NCHAR
- issues, *7-33*

NCLOB
- data type, *5-26*

nested table
- direct path loading of, *21-17*
- element ordering, *20-21*
- functions for manipulating, *20-21*

new release, relinking, *3-20*

NLS_LANG, *4-40*

NLS_NCHAR, *4-40*

NOCOMPRESS, *9-28*

non-deferred linkage no longer supported, *10-34*

nonblocking mode, *4-36*

nonfinal object tables
- direct path loading of, *21-31*

notification descriptor attributes, *A-96*

NULL indicator
- setting for an object attribute, *19-28*

NULL indicator struct, *19-28*
- generated by OTT, *19-8*

nullness
- of objects, *19-28*

NULLs
- atomic, *19-28*
- inserting, *4-32*
- inserting into database, *4-31*
- inserting using indicator variables, *4-31*

NUMBER
- external data type, *5-12*

## O

object applications
  commit, *22-15*
  database connection, *19-10*
  rollback, *22-15*
object cache, *22-1*
  coherency, *22-4*
  consistency, *22-4*
  initializing, *19-9*
  loading objects, *22-6*
  memory parameters, *22-4*
  operations on, *22-5*
  removing objects, *22-6*
  setting the size of, *22-4*
object functions
  server round-trips, *C-5*
object identifiers, *19-4*, *19-33*, *19-34*
  for persistent objects, *19-4*
object references, *19-10*, *19-33*
    *See also* REFs
object run-time environment
  initializing, *19-9*
object tables
  direct path loading of, *21-30*
object type, *19-8*
  representing in C applications, *19-8*
Object Type Translator utility
  *See* OTT utility
object view, *19-17*
objects
  accessing with OCI, *24-20*
  allocation duration, *22-15*
  array pin, *19-13*
  attributes, *19-17*
    manipulating, *19-13*
  client-side cache, *22-1*
  copying, *19-31*
  creating, *19-31*
  duration, *22-15*
  flushing, *22-11*
  flushing changes, *19-15*
  freeing, *19-31*, *22-9*
  lifetime, *27-2*
  LOB attributes of transient objects, *9-4*
  locking, *22-13*
  manipulating with OCI, *24-20*
  marking, *19-15*, *22-10*
  memory layout of instance, *22-17*
  memory management, *22-1*
  meta-attributes, *19-17*
  navigation, *22-18*
    simple, *22-18*
  NCHAR and NVARCHAR2 attribute of, *20-2*
  NULL values, *19-28*

objects *(continued)*
  OCI object application structure, *19-3*
  persistent, *19-4*
  pin count, *19-28*
  pin duration, *22-15*
  pinning, *19-11*, *22-7*
  refreshing, *22-12*
  secondary memory, *22-17*
  terminology, *19-4*, *27-2*
  top-level memory, *22-17*
  transient, *19-4*, *19-5*
  types, *19-4*, *27-2*
  unmarking, *22-10*
  unpinning, *22-8*
  use with OCI, *19-1*
OCCI
  alternative to OCI, *1-3*
OCI, *1-1*
  aborting calls, *4-33*
  accessing and manipulating objects, *24-20*
  advantages, *1-1*
  object support, *1-10*
  Oracle XA Library, *D-5*
  overview, *1-1*
  parts of, *1-1*
  sample programs, *D-2*
OCI applications, *4-1*
  compiling, *1-3*, *D-2*
  general structure, *4-1*
  initialization example, *4-23*
  linking, *1-3*, *D-3*
  running, *D-4*
  steps, *4-17*
  structure, *4-1*
  structure using objects, *19-3*
  terminating, *4-26*
  using the OTT utility with, *24-19*
  with objects
    initializing, *19-9*
OCI demonstration programs
  demo subdirectory, *3-2*
OCI environment
  initializing for objects, *19-9*
OCI functions
  canceling calls, *4-33*
  data cartridges, *4-1*
  globalization, *4-1*
  not supported, *3-24*
  obsolescent, *3-22*
  return codes, *4-27*, *4-29*
OCI handle types, *4-3*
OCI interface
  XStream, *34-1*
OCI navigational functions, *22-21*
  flush functions, *22-22*

OCI navigational functions *(continued)*
  mark functions, *22-22*
  meta-attribute accessor functions, *22-23*
  miscellaneous functions, *22-23*
  naming scheme, *22-21*
  pin/unpin/free functions, *22-22*
OCI process
  initializing for objects, *19-9*
OCI program
  *See* OCI applications
OCI relational functions
  connect, authorize, and initialize, *25-3*
  guide to reference entries, *29-1*
  Oracle Database Advanced Queuing and publish-subscribe, *26-79*
  Oracle Streams Advanced Queuing and publish-subscribe, *E-27*
OCI session pool, *18-5*
OCI session pool configuration
  using oraaccess.xml, *3-17*
OCI SODA functions, *36-1*
OCI support for implicit results, *15-9*
OCI_ATTR_ACCESS_BANNER, *10-33*
  server handle attribute, *A-15*
OCI_ATTR_ACTION, *10-25*
  user session handle attribute, *A-21*
OCI_ATTR_ADMIN_PFILE
  administration handle attribute, *A-34*
OCI_ATTR_AGENT_ADDRESS
  OCIAQAgent descriptor attribute, *A-85*
OCI_ATTR_AGENT_NAME
  OCIAQAgent descriptor attribute, *A-85*
OCI_ATTR_AGENT_PROTOCOL
  OCIAQAgent descriptor attribute, *A-85*
OCI_ATTR_ALLOC_DURATION
  environment handle attribute, *A-3*
OCI_ATTR_APPCTX_ATTR, *10-24*
  user session handle attribute, *A-21*
OCI_ATTR_APPCTX_LIST, *10-23*
  user session handle attribute, *A-21*
OCI_ATTR_APPCTX_NAME, *10-24*
  user session handle attribute, *A-21*
OCI_ATTR_APPCTX_SIZE, *10-23*, *25-32*
  user session handle attribute, *A-21*
OCI_ATTR_APPCTX_VALUE, *10-24*
  user session handle attribute, *A-21*
OCI_ATTR_AQ_NTFN_GROUPING_MSGID_ARRAY
  notification descriptor attribute, *A-96*
OCI_ATTR_AQ_NTFN_GROUPING_COUNT
  notification descriptor attribute, *A-96*
OCI_ATTR_ATTEMPTS
  OCIAQMsgProperties descriptor attribute, *A-81*
OCI_ATTR_AUDIT_BANNER
  user session handle attribute, *A-21*
OCI_ATTR_AUTOCOMMIT_DDL
  attribute, *8-23*
OCI_ATTR_AUTOTUNING_ENABLED
  user session handle attribute, *A-21*
OCI_ATTR_BIND_COUNT
  statement handle attribute, *A-41*
OCI_ATTR_BIND_DN
  environment handle attribute, *A-3*
OCI_ATTR_BREAK_ON_NET_TIMEOUT
  server handle attribute, *A-15*
OCI_ATTR_BUF_ADDR
  direct path loading stream handle attribute, *A-109*
OCI_ATTR_BUF_SIZE
  direct path loading
    context handle attribute, *A-99*
    stream handle attribute, *A-109*
OCI_ATTR_CACHE
  attribute, *8-15*
OCI_ATTR_CACHE_ARRAYFLUSH, *22-11*
  environment handle attribute, *A-3*
OCI_ATTR_CACHE_MAX_SIZE, *22-4*
  environment handle attribute, *A-3*
OCI_ATTR_CACHE_OPT_SIZE, *22-4*
  environment handle attribute, *A-3*
OCI_ATTR_CALL_TIME, *10-25*
  user session handle attribute, *A-21*
OCI_ATTR_CALL_TIMEOUT
  service contect handle attribute, *A-10*
OCI_ATTR_CATALOG_LOCATION
  attribute, *8-23*
OCI_ATTR_CERTIFICATE, *10-17*, *25-32*
  user session handle attribute, *A-21*
OCI_ATTR_CHAR_COUNT
  bind handle attribute, *A-51*
  define handle attribute, *A-54*
OCI_ATTR_CHAR_SIZE, *8-16*
  attribute, *8-26*, *8-27*
OCI_ATTR_CHAR_USED, *8-16*
  attribute, *8-26*
OCI_ATTR_CHARSET_FORM, *7-35*
  attribute, *8-11*, *8-13*, *8-16*, *8-20*
  bind handle attribute, *A-51*
  define handle attribute, *A-54*
OCI_ATTR_CHARSET_ID
  attribute, *8-11*, *8-13*, *8-16*, *8-20*, *8-23*
  bind handle attribute, *A-51*
  define handle attribute, *A-54*
  direct path loading
    column parameter attribute, *A-110*
    context handle attribute, *A-99*

OCI_ATTR_CHDES_DBNAME
    continuous query notification descriptor attribute, *A-93*
OCI_ATTR_CHDES_NFTYPE
    continuous query notification descriptor attribute, *A-93*
OCI_ATTR_CHDES_ROW_OPFLAGS
    continuous query notification descriptor attribute, *A-93*
OCI_ATTR_CHDES_ROW_ROWID
    continuous query notification descriptor attribute, *A-93*
OCI_ATTR_CHDES_TABLE_CHANGES
    continuous query notification descriptor attribute, *A-93*
OCI_ATTR_CHDES_TABLE_NAME
    continuous query notification descriptor attribute, *A-93*
OCI_ATTR_CHDES_TABLE_OPFLAGS
    continuous query notification descriptor attribute, *A-93*
OCI_ATTR_CHDES_TABLE_ROW_CHANGES
    continuous query notification descriptor attribute, *A-93*
OCI_ATTR_CHNF_CHANGELAG
    continuous query notification attribute, *A-92*
OCI_ATTR_CHNF_OPERATIONS
    continuous query notification attribute, *A-92*
OCI_ATTR_CHNF_REGHANDLE
    statement handle attribute, *A-41*
OCI_ATTR_CHNF_ROWIDS
    continuous query notification attribute, *A-92*
OCI_ATTR_CHNF_TABLENAMES
    continuous query notification attribute, *A-92*
OCI_ATTR_CLIENT_IDENTIFIER, *10-18*
    user session handle attribute, *A-21*
OCI_ATTR_CLIENT_INFO, *10-25*, *25-32*
    user session handle attribute, *A-21*
OCI_ATTR_CLUSTERED
    attribute, *8-8*
OCI_ATTR_COL_COUNT
    direct path loading column array handle attribute, *A-108*
OCI_ATTR_COL_PROPERTIES, *8-16*
OCI_ATTR_COLLATION_ID, *8-16*
    bind handle attribute, *A-51*
OCI_ATTR_COLLECT_CALL_TIME, *10-25*
    user session handle attribute, *A-21*
OCI_ATTR_COLLECTION_ELEMENT
    attribute, *8-9*
OCI_ATTR_COLLECTION_TYPECODE
    attribute, *8-9*
OCI_ATTR_COMMENT
    attribute, *8-24*
OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE
    COR handle attribute, *A-74*
OCI_ATTR_COMPLEXOBJECT_LEVEL
    COR handle attribute, *A-74*
OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL
    COR descriptor attribute, *A-74*
OCI_ATTR_COMPLEXOBJECTCOMP_TYPE
    COR descriptor attribute, *A-74*
OCI_ATTR_CONDITION
    attribute, *8-24*
OCI_ATTR_CONN_BUSY_COUNT
    connection pool handle attribute, *A-34*
OCI_ATTR_CONN_INCR
    connection pool handle attribute, *A-34*
OCI_ATTR_CONN_MAX
    connection pool handle attribute, *A-34*
OCI_ATTR_CONN_MIN
    connection pool handle attribute, *A-34*
OCI_ATTR_CONN_NOWAIT
    connection pool handle attribute, *A-34*
OCI_ATTR_CONN_OPEN_COUNT
    connection pool handle attribute, *A-34*
OCI_ATTR_CONN_TIMEOUT
    connection pool handle attribute, *A-34*
OCI_ATTR_CONNECTION_CLASS
    user session handle attribute, *A-21*
OCI_ATTR_CONSUMER_NAME, *A-77*
    notification descriptor attribute, *A-96*
OCI_ATTR_CORRELATION
    OCIAQDeqOptions descriptor attribute, *A-77*
    OCIAQMsgProperties descriptor attribute, *A-81*
OCI_ATTR_CQ_QUERYID
    statement handle attribute, *A-41*
OCI_ATTR_CQDES_OPERATION
    direct path loading handle attribute, *A-97*
OCI_ATTR_CQDES_QUERYID
    direct path loading handle attribute, *A-97*
OCI_ATTR_CQDES_TABLE_CHANGES
    direct path loading handle attribute, *A-97*
OCI_ATTR_CURRENT_POSITION
    attribute, *6-19*
    statement handle attribute, *A-41*
OCI_ATTR_CURRENT_SCHEMA, *17-3*, *25-32*
    user session handle attribute, *A-21*
OCI_ATTR_CURSOR_COMMIT_ BEHAVIOR
    attribute, *8-23*
OCI_ATTR_DATA_SIZE, *8-26*
    attribute, *8-11*, *8-13*, *8-16*, *8-20*, *8-27*
    direct path loading column parameter attribute, *A-110*
OCI_ATTR_DATA_TYPE
    attribute, *8-3*, *8-11*, *8-13*, *8-16*, *8-20*

OCI_ATTR_DATA_TYPE *(continued)*
  direct path loading column parameter attribute, *A-110*
OCI_ATTR_DATEFORMAT
  direct path loading
    column parameter attribute, *A-110*
    context handle attribute, *A-99*
OCI_ATTR_DBDOMAIN
  event handle attribute, *A-116*
OCI_ATTR_DBNAME
  event handle attribute, *A-116*
OCI_ATTR_DBOP, *10-25*, *25-32*
  user session handle attribute, *A-21*
OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE
  user session handle attribute, *A-21*
OCI_ATTR_DELAY
  OCIAQMsgProperties descriptor attribute, *A-81*
OCI_ATTR_DEQ_MODE
  OCIAQDeqOptions descriptor attribute, *A-77*
OCI_ATTR_DEQ_MSGID
  OCIAQDeqOptions descriptor attribute, *A-77*
OCI_ATTR_DEQCOND
  OCIAQDeqOptions descriptor attribute, *A-77*
OCI_ATTR_DESC_PUBLIC, *25-103*
OCI_ATTR_DIRPATH_DCACHE_DISABLE, *21-15*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_DCACHE_HITS, *21-15*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_DCACHE_MISSES, *21-15*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_DCACHE_NUM, *21-15*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_DCACHE_SIZE, *21-15*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_DEF_EXP_CACHE_SIZE
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_DEFAULTS
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_EXPR_TYPE
  direct path function context handle attribute, *A-106*
  function context attribute, *21-36*
OCI_ATTR_DIRPATH_FLAGS
  direct path loading context handle attribute, *21-16*, *A-99*
OCI_ATTR_DIRPATH_INDEX_MAINT_METHOD
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_MODE
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_NO_INDEX_ERRORS, *21-37*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_NOLOG
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_OBJ_CONSTR, *21-35*
  direct path context attribute, *21-34*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_OID
  direct path loading column parameter attribute, *A-110*
OCI_ATTR_DIRPATH_PARALLEL, *21-1*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_PGA_LIM
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_REJECT_ROWS_REPCH
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DIRPATH_SID
  column array attribute, *21-41*
  direct path loading column parameter attribute, *A-110*
OCI_ATTR_DIRPATH_SKIPINDEX_METHOD
  direct path loading context handle attribute, *A-99*
OCI_ATTR_DISTINGUISHED_NAME, *10-17*
  user session handle attribute, *A-21*
OCI_ATTR_DML_ROW_COUNT_ARRAY
  statement handle attribute, *A-41*
OCI_ATTR_DML_ROW_OFFSET
  error handle attribute, *A-10*
OCI_ATTR_DN_COUNT
  OCIServerDNs descriptor attribute, *A-85*
OCI_ATTR_DRIVER_NAME, *25-32*
  user session handle attribute, *A-21*
OCI_ATTR_DURATION
  attribute, *8-7*
OCI_ATTR_EDITION, *17-3*
  user session handle attribute, *A-21*
OCI_ATTR_ENCAPSULATION
  attribute, *8-12*
OCI_ATTR_ENQ_TIME
  OCIAQMsgProperties descriptor attribute, *A-81*

OCI_ATTR_ENV
  server handle attribute, *A-15*
  service context handle attribute, *A-10*
  statement handle attribute, *A-41*
OCI_ATTR_ENV_CHARSET_ID
  environment handle attribute, *A-3*
OCI_ATTR_ENV_NCHARSET_ID, *4-40*
  environment handle attribute, *A-3*
OCI_ATTR_ENV_NLS_LANGUAGE
  environment handle attribute, *4-41*, *A-3*
OCI_ATTR_ENV_NLS_TERRITORY, *4-41*
  environment handle attribute, *A-3*
OCI_ATTR_ENV_UTF16
  environment handle attribute, *A-3*
OCI_ATTR_EVAL_CONTEXT_NAME
  attribute, *8-24*
OCI_ATTR_EVAL_CONTEXT_OWNER
  attribute, *8-24*
OCI_ATTR_EVALUATION_FUNCTION
  attribute, *8-24*
OCI_ATTR_EVENTTYPE
  event handle attribute, *A-116*
OCI_ATTR_EVTCBK, *12-4*
  environment handle attribute, *A-3*
OCI_ATTR_EVTCTX, *12-4*
  environment handle attribute, *A-3*
OCI_ATTR_EXCEPTION_QUEUE
  OCIAQMsgProperties descriptor attribute, *A-81*
OCI_ATTR_EXPIRATION
  OCIAQMsgProperties descriptor attribute, *A-81*
OCI_ATTR_EXTERNAL_NAME, *10-6*
  server handle attribute, *A-15*
OCI_ATTR_FETCH_ROWID
  statement handle attribute, *A-41*
OCI_ATTR_FIXUP_CALLBACK
  authentication handle attribute, *11-4*, *A-20*
OCI_ATTR_FOCBK
  server handle attribute, *A-15*
  session pool handle attribute, *A-36*
OCI_ATTR_FSPRECISION
  attribute, *8-11*
OCI_ATTR_HA_SOURCE
  event handle attribute, *A-116*
OCI_ATTR_HA_SRVFIRST
  event handle attribute, *A-116*
OCI_ATTR_HA_SRVNEXT, *12-4*
  event handle attribute, *A-116*
OCI_ATTR_HA_STATUS
  event handle attribute, *A-116*
OCI_ATTR_HA_TIMESTAMP
  event handle attribute, *A-116*
OCI_ATTR_HAS_DEFAULT
  attribute, *8-20*
OCI_ATTR_HAS_FILE
  attribute, *8-9*
OCI_ATTR_HAS_LOB
  attribute, *8-9*
OCI_ATTR_HAS_NESTED_TABLE
  attribute, *8-9*
OCI_ATTR_HEAPALLOC
  environment handle attribute, *A-3*
OCI_ATTR_HOSTNAME
  event handle attribute, *A-116*
OCI_ATTR_HW_MARK
  attribute, *8-15*
OCI_ATTR_IMPLICIT_RSET_COUNT, *A-41*
OCI_ATTR_IN_V8_MODE
  server handle attribute, *A-15*
  service context handle attribute, *A-10*
OCI_ATTR_INCR
  attribute, *8-15*
OCI_ATTR_INDEX_ONLY
  attribute, *8-8*
OCI_ATTR_INITIAL_CLIENT_ROLES, *10-18*, *17-3*
  user session handle attribute, *A-21*
OCI_ATTR_INSTNAME, *18-7*
  event handle attribute, *A-116*
  server handle attribute, *A-15*
  service context handle attribute, *A-10*
  shard instance descriptor attribute, *A-59*
OCI_ATTR_INSTSTARTTIME
  event handle attribute, *A-116*
OCI_ATTR_INTERNAL_NAME
  server handle attribute, *A-15*
OCI_ATTR_IOMODE
  attribute, *8-20*
OCI_ATTR_IS_CONSTRUCTOR
  attribute, *8-12*
OCI_ATTR_IS_DESTRUCTOR
  attribute, *8-12*
OCI_ATTR_IS_FINAL_METHOD
  attribute, *8-12*
OCI_ATTR_IS_FINAL_TYPE
  attribute, *8-9*
OCI_ATTR_IS_INCOMPLETE_TYPE
  attribute, *8-9*
OCI_ATTR_IS_INSTANTIABLE_METHOD
  attribute, *8-12*
OCI_ATTR_IS_INSTANTIABLE_TYPE
  attribute, *8-9*
OCI_ATTR_IS_INVOKER_RIGHTS
  attribute, *8-8*, *8-9*
OCI_ATTR_IS_MAP
  attribute, *8-12*
OCI_ATTR_IS_NULL
  attribute, *8-16*, *8-20*

OCI_ATTR_IS_OPERATOR
  attribute, *8-12*
OCI_ATTR_IS_ORDER
  attribute, *8-12*
OCI_ATTR_IS_OVERRIDING_METHOD
  attribute, *8-12*
OCI_ATTR_IS_PREDEFINED_TYPE
  attribute, *8-9*
OCI_ATTR_IS_RECOVERABLE, *A-10*
OCI_ATTR_IS_RNDS
  attribute, *8-12*
OCI_ATTR_IS_RNPS
  attribute, *8-12*
OCI_ATTR_IS_SELFISH
  attribute, *8-12*
OCI_ATTR_IS_SUBTYPE
  attribute, *8-9*
OCI_ATTR_IS_SYSTEM_GENERATED_TYPE
  attribute, *8-9*
OCI_ATTR_IS_SYSTEM_TYPE
  attribute, *8-9*
OCI_ATTR_IS_TEMPORARY
  attribute, *8-7*
OCI_ATTR_IS_TRANSIENT_TYPE
  attribute, *8-9*
OCI_ATTR_IS_WNDS
  attribute, *8-12*
OCI_ATTR_IS_WNPS
  attribute, *8-12*
OCI_ATTR_LDAP_AUTH
  environment handle attribute, *A-3*
OCI_ATTR_LDAP_CRED
  environment handle attribute, *A-3*
OCI_ATTR_LDAP_CTX
  environment handle attribute, *A-3*
OCI_ATTR_LDAP_HOST
  environment handle attribute, *A-3*
OCI_ATTR_LDAP_PORT
  environment handle attribute, *A-3*
OCI_ATTR_LEVEL
  attribute, *8-20*
OCI_ATTR_LFPRECISION
  attribute, *8-11*
OCI_ATTR_LINK
  attribute, *8-14*, *8-20*
OCI_ATTR_LIST_ACTION_CONTEXT
  attribute, *8-24*
OCI_ATTR_LIST_ARGUMENTS
  attribute, *8-5*, *8-8*, *8-12*
OCI_ATTR_LIST_COLUMNS
  attribute, *8-7*
  direct path loading
    context handle attribute, *A-99*
    function context handle attribute, *A-106*
OCI_ATTR_LIST_OBJECTS
  attribute, *8-22*
OCI_ATTR_LIST_RULES
  attribute, *8-24*
OCI_ATTR_LIST_SCHEMAS
  attribute, *8-23*
OCI_ATTR_LIST_SUBPROGRAMS
  attribute, *8-9*
OCI_ATTR_LIST_TABLE_ALIASES
  attribute, *8-24*
OCI_ATTR_LIST_TYPE
  attribute, *8-22*
OCI_ATTR_LIST_TYPE_ATTRS
  attribute, *8-9*
OCI_ATTR_LIST_TYPE_METHODS
  attribute, *8-9*
OCI_ATTR_LIST_VARIABLE_TYPES
  attribute, *8-24*
OCI_ATTR_LOB_REMOTE
  LOB locator attribute, *A-73*
OCI_ATTR_LOB_TYPE
  LOB descriptor attribute, *A-73*
OCI_ATTR_LOBEMPTY
  LOB locator attribute, *A-73*
OCI_ATTR_LOBPREFETCH_LENGTH, *9-25*
  define handle attribute, *A-54*
OCI_ATTR_LOBPREFETCH_SIZE
  define handle attribute, *A-54*
OCI_ATTR_LOCKING_MODE
  attribute, *8-23*
OCI_ATTR_LTXID, *A-21*
OCI_ATTR_MAP_METHOD
  attribute, *8-9*
OCI_ATTR_MAX
  attribute, *8-15*
OCI_ATTR_MAX_CATALOG_ NAMELEN
  attribute, *8-23*
OCI_ATTR_MAX_CHARSET_RATIO
  service context handle attribute, *A-10*
OCI_ATTR_MAX_COLUMN_ NAMELEN
  attribute, *8-23*
OCI_ATTR_MAX_NCHARSET_RATIO
  service context handle attribute, *A-10*
OCI_ATTR_MAX_OPEN_CURSORS, *17-3*, *A-21*
OCI_ATTR_MAX_PROC_LEN
  attribute, *8-23*
OCI_ATTR_MAXCHAR_SIZE
  attribute, *7-36*, *7-38*
  bind handle attribute, *A-51*
  define handle attribute, *A-54*
OCI_ATTR_MAXDATA_SIZE
  attribute, *7-36*
  bind handle attribute, *A-51*
  use with binding, *7-37*

OCI_ATTR_MEMPOOL_APPNAME
  process handle attribute, *A-114*
OCI_ATTR_MEMPOOL_HOMENAME
  process handle attribute, *A-114*
OCI_ATTR_MEMPOOL_INSTNAME
  process handle attribute, *A-114*
OCI_ATTR_MEMPOOL_SIZE
  process handle attribute, *A-114*
OCI_ATTR_MIGSESSION
  user session handle attribute, *A-21*
OCI_ATTR_MIN
  attribute, *8-15*
OCI_ATTR_MODULE, *10-25*
  user session handle attribute, *A-21*
OCI_ATTR_MSG_DELIVERY_MODE, *26-79*, *26-82*
  OCIAQDeqOptions descriptor attribute, *A-77*
  OCIAQEnqOptions descriptor attribute, *A-75*
  OCIAQMsgProperties descriptor attribute, *A-81*
OCI_ATTR_MSG_PROP
  notification descriptor attribute, *A-96*
OCI_ATTR_MSG_STATE
  OCIAQMsgProperties descriptor attribute, *A-81*
OCI_ATTR_NAME
  attribute, *8-8*, *8-9*, *8-11*–*8-14*, *8-16*, *8-20*, *8-25*, *8-26*
  column array attribute, *21-39*
  direct path loading
    column parameter attribute, *A-110*
    context handle attribute, *A-99*
    function context attribute, *21-35*
    function context handle attribute, *A-106*
OCI_ATTR_NAVIGATION
  OCIAQDeqOptions descriptor attribute, *A-77*
OCI_ATTR_NCHARSET_ID
  attribute, *8-23*
OCI_ATTR_NFY_FLAGS
  notification descriptor attribute, *A-96*
OCI_ATTR_NFY_MSGID
  notification descriptor attribute, *A-96*
OCI_ATTR_NONBLOCKING_MODE
  server handle attribute, *4-37*, *A-15*
OCI_ATTR_NOWAIT_SUPPORT
  attribute, *8-23*
OCI_ATTR_NUM_COLS
  attribute, *8-7*
  direct path loading
    column array handle attribute, *A-108*
    function context attribute, *21-37*
    function context handle attribute, *A-106*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_NUM_DML_ERRORS
  statement handle attribute, *A-41*
OCI_ATTR_NUM_ELEMS
  attribute, *8-13*
OCI_ATTR_NUM_HANDLES attribute, *8-22*
OCI_ATTR_NUM_ROWS
  attribute, *21-42*
  direct path loading
    column array handle attribute, *A-108*
    context handle attribute, *A-99*
    function context handle attribute, *A-106*
  function context attribute, *21-38*
OCI_ATTR_NUM_TYPE_ATTRS
  attribute, *8-9*
OCI_ATTR_NUM_TYPE_METHODS
  attribute, *8-9*
OCI_ATTR_OBJ_ID
  attribute, *8-5*
OCI_ATTR_OBJ_NAME
  attribute, *8-5*
OCI_ATTR_OBJ_SCHEMA
  attribute, *8-5*
OCI_ATTR_OBJECT
  environment handle attribute, *A-3*
OCI_ATTR_OBJECT_DETECTCHANGE
  environment handle attribute, *22-14*, *A-3*
OCI_ATTR_OBJECT_NEWNOTNULL, *27-36*
  environment handle attribute, *A-3*
OCI_ATTR_OBJID
  attribute, *8-7*, *8-14*, *8-15*
OCI_ATTR_ORA_DBUG_JDWP
  user session handle attribute, *A-21*
OCI_ATTR_ORDER
  attribute, *8-15*
OCI_ATTR_ORDER_METHOD
  attribute, *8-9*
OCI_ATTR_ORIGINAL_MSGID
  OCIAQMsgProperties descriptor attribute, *A-81*
OCI_ATTR_PARAM
  describe handle attribute, *A-57*
  use when an attribute is itself a descriptor, *25-48*
OCI_ATTR_PARAM_COUNT
  describe handle attribute, *A-57*
  statement handle attribute, *A-41*
OCI_ATTR_PARSE_ERROR_OFFSET
    statement handle attribute, *A-41*
OCI_ATTR_PARTITIONED
  attribute, *8-8*
OCI_ATTR_PASSWORD, *10-19*
  user session handle attribute, *A-21*
OCI_ATTR_PDPRC
  bind handle attribute, *A-51*
  define handle attribute, *A-54*

- OCI_ATTR_PDSCL
  - bind handle attribute, *A-51*
  - define handle attribute, *A-54*
- OCI_ATTR_PIN_DURATION
  - environment handle attribute, *A-3*
- OCI_ATTR_PINOPTION
  - environment handle attribute, *A-3*
- OCI_ATTR_POSITION
  - attribute, *8-20*
- OCI_ATTR_PRECISION
  - attribute, *8-4*, *8-11*, *8-13*, *8-16*, *8-20*
  - direct path loading column parameter attribute, *A-110*
- OCI_ATTR_PREFETCH_MEMORY
  - statement handle attribute, *A-41*
- OCI_ATTR_PREFETCH_ROWS
  - statement handle attribute, *A-41*
- OCI_ATTR_PRIORITY
  - OCIAQMsgProperties descriptor attribute, *A-81*
- OCI_ATTR_PROC_MODE
  - process handle attribute, *A-114*
- OCI_ATTR_PROXY_CLIENT
  - user session handle attribute, *A-21*
- OCI_ATTR_PROXY_CREDENTIALS, *10-17*
  - user session handle attribute, *A-21*
- OCI_ATTR_PTYPE
  - attribute, *8-5*
  - possible values of, *8-5*
- OCI_ATTR_PURITY, *25-32*
  - user session handle attribute, *A-21*
- OCI_ATTR_QUEUE_NAME
  - notification descriptor attribute, *A-96*
- OCI_ATTR_RADIX
  - attribute, *8-20*
- OCI_ATTR_RDBA
  - attribute, *8-8*
- OCI_ATTR_RECEIVE_TIMEOUT
  - server handle attribute, *A-15*
- OCI_ATTR_RECIPIENT_LIST
  - OCIAQMsgProperties descriptor attribute, *A-81*
- OCI_ATTR_REF_TDO
  - attribute, *8-7*, *8-9*, *8-11*, *8-13*, *8-16*, *8-20*
- OCI_ATTR_RELATIVE_MSGID
  - OCIAQEnqOptions descriptor attribute, *A-75*
- OCI_ATTR_ROW_COUNT, *6-19*
  - direct path loading
    - column array handle attribute, *A-108*
    - stream handle attribute, *A-109*
  - statement handle attribute, *A-41*
- OCI_ATTR_ROWID
  - statement handle attribute, *A-41*
- OCI_ATTR_ROWS_FETCHED, *6-19*
  - statement handle attribute, *A-41*
- OCI_ATTR_ROWS_RETURNED
  - bind handle attribute, *A-51*
  - use with callbacks, *7-33*
- OCI_ATTR_SAVEPOINT_SUPPORT
  - attribute, *8-23*
- OCI_ATTR_SCALE
  - attribute, *8-11*, *8-13*, *8-16*, *8-20*
  - direct path loading column parameter attribute, *A-110*
- OCI_ATTR_SCHEMA_NAME
  - attribute, *8-9*, *8-11*, *8-13*, *8-14*, *8-16*, *8-20*
  - direct path loading context handle attribute, *A-99*
- OCI_ATTR_SEND_TIMEOUT
  - server handle attribute, *A-15*
- OCI_ATTR_SENDER_ID
  - OCIAQMsgProperties descriptor attribute, *A-81*
- OCI_ATTR_SEQ
  - attributes, *8-15*
- OCI_ATTR_SEQUENCE enqueue option, *13-25*
- OCI_ATTR_SEQUENCE_DEVIATION
  - OCIAQEnqOptions descriptor attribute, *A-75*
- OCI_ATTR_SERVER
  - service context handle attribute, *A-10*
- OCI_ATTR_SERVER_DN
  - OCIServerDNs descriptor attribute, *A-85*
- OCI_ATTR_SERVER_DNS
  - subscription handle attribute, *A-86*
- OCI_ATTR_SERVER_GROUP, *10-14*
  - server handle attribute, *A-15*
- OCI_ATTR_SERVER_STATUS, *4-27*
  - server handle attribute, *A-15*
- OCI_ATTR_SERVICENAME
  - event handle attribute, *A-116*
- OCI_ATTR_SESSION
  - service context handle attribute, *A-10*
- OCI_ATTR_SESSION_STATE
  - user session handle attribute, *A-21*
- OCI_ATTR_SHARD_HAS_WRITABLECHUNK
  - shard instance descriptor attribute, *A-59*
- OCI_ATTR_SHARDING_KEY, *18-1*, *18-5*, *18-7*
  - service context handle attribute, *A-21*
- OCI_ATTR_SHARDING_KEY_64
  - service context handle attribute, *A-21*
- OCI_ATTR_SHARED_HEAPALLOC
  - environment handle attribute, *A-3*
- OCI_ATTR_SHOW_INVISIBLE_COLUMNS, *A-57*
  - attribute, *8-36*
- OCI_ATTR_SODA_COLL_NAME
  - SODA collection handle attribute, *A-62*
- OCI_ATTR_SODA_CONTENT
  - SODA document handle attributes, *A-60*

OCI_ATTR_SODA_CREATE_TIMESTAMP
  SODA document handle attributes, *A-60*
OCI_ATTR_SODA_CRTIME_COL_NAME
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_CTNT_CACHE
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_CTNT_COL_NAME
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_CTNT_COMPRESS
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_CTNT_ENCRYPT
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_CTNT_MAX_LEN
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_CTNT_SQL_TYPE
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_CTNT_VALIDATION
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_DESCRIPTOR
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_DETECT_JSON_ENC
  SODA document handle attributes, *A-60*
OCI_ATTR_SODA_JSON_CHARSET_ID
  SODA document handle attributes, *A-60*
OCI_ATTR_SODA_KEY
  SODA document handle attributes, *A-60*
OCI_ATTR_SODA_KEY_ASSIGN_METHOD
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_KEY_COL_NAME
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_KEY_MAX_LEN
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_KEY_SEQ_NAME
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_KEY_SQL_TYPE
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_LASTMOD_TIMESTAMP
  SODA document handle attributes, *A-60*
OCI_ATTR_SODA_MEDIA_TYPE
  SODA document handle attributes, *A-60*
OCI_ATTR_SODA_MODTIME_COL_NAME
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_MODTIME_INDEX
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_MTYPE_COL_NAME
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_READONLY
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_SCHEMA
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_TABLE_NAME
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_VERSION
  SODA document handle attributes, *A-60*
OCI_ATTR_SODA_VERSION_COL_NAME
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_VERSION_METHOD
  SODA collection handle attribute, *A-62*
OCI_ATTR_SODA_VIEW_NAME
  SODA collection handle attribute, *A-62*
OCI_ATTR_SPOOL_AUTH
  session pool handle attribute, *11-4*, *A-36*
OCI_ATTR_SPOOL_BUSY_COUNT
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_GETMODE
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_INCR
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_MAX
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_MAX_LIFETIME_SESSION
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_MAX_USE_SESSION
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_MIN
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_OPEN_COUNT
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_STMTCACHESIZE
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_TIMEOUT
  session pool handle attribute, *A-36*
OCI_ATTR_SPOOL_WAIT_TIMEOUT
  session pool handle attribute, *A-36*
OCI_ATTR_SQL_ID
  statement handle attribute, *A-41*
OCI_ATTR_SQLFNCODE
  statement handle attribute, *A-41*
OCI_ATTR_STATEMENT
  statement handle attribute, *A-41*
OCI_ATTR_STMT_STATE
  statement handle attribute, *A-41*
OCI_ATTR_STMT_TYPE
  statement handle attribute, *A-41*
OCI_ATTR_STMTCACHE_CBK
  service context handle attribute, *A-10*
OCI_ATTR_STMTCACHE_CBKCTX
  statement handle attribute, *A-41*
OCI_ATTR_STMTCACHESIZE, *15-3*, *25-28*
  service context handle attribute, *A-10*
OCI_ATTR_STREAM_OFFSET
  direct path loading stream handle attribute, *A-109*
OCI_ATTR_SUB_NAME
  attribute, *8-20*
  direct path loading context handle attribute, *A-99*
OCI_ATTR_SUBSCR_CALLBACK
  subscription handle attribute, *A-86*

OCI_ATTR_SUBSCR_CQ_QOSFLAGS
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_CTX
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_HOSTADDR
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_IPADDR
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_NAME
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_NAMESPACE, *13-4*, *13-9*
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_NTFN_GROUPING_CLASS, *A-86*
OCI_ATTR_SUBSCR_NTFN_GROUPING_REPEAT_COUNT
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_NTFN_GROUPING_START_TIME
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_NTFN_GROUPING_TYPE
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_NTFN_GROUPING_VALUE
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_PAYLOAD
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_PORTNO
environment handle attribute, *A-3*
OCI_ATTR_SUBSCR_QOSFLAGS
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_RECPT
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_RECPTPRES
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_RECPTPROTO
subscription handle attribute, *A-86*
OCI_ATTR_SUBSCR_SERVER_DN
descriptor handle, *13-9*
OCI_ATTR_SUBSCR_TIMEOUT
subscription handle attribute, *A-86*
OCI_ATTR_SUPER_SHARDING_KEY, *18-1*, *18-7*
service context handle attribute, *A-21*
OCI_ATTR_SUPER_SHARDINGING_KEY, *18-5*
OCI_ATTR_SUPERTYPE_NAME
attribute, *8-9*
OCI_ATTR_SUPERTYPE_SCHEMA_NAME
attribute, *8-9*
OCI_ATTR_TABLE_NAME
attribute, *8-25*
OCI_ATTR_TABLESPACE
attribute, *8-8*
OCI_ATTR_TAF_ENABLED
server handle attribute, *A-15*
OCI_ATTR_TIMESTAMP
attribute, *8-5*
OCI_ATTR_TRANS
service context handle attribute, *A-10*
OCI_ATTR_TRANS_NAME, *10-3*
transaction handle attribute, *A-40*
OCI_ATTR_TRANS_PROFILE_FOREIGN
user session handle attribute, *A-21*
OCI_ATTR_TRANS_TIMEOUT
transaction handle attribute, *A-40*
OCI_ATTR_TRANSACTION_IN_PROGRESS
user session handle attribute, *A-21*
OCI_ATTR_TRANSACTION_NO
OCIAQMsgProperties descriptor attribute, *A-81*
OCI_ATTR_TRANSFORMATION (dequeue)
OCIAQDeqOptions descriptor attribute, *A-77*
OCI_ATTR_TRANSFORMATION (enqueue)
OCIAQEnqOptions descriptor attribute, *A-75*
OCI_ATTR_TYPE_NAME
attribute, *8-11*, *8-13*, *8-16*, *8-20*
OCI_ATTR_TYPECODE
attribute, *8-3*, *8-9*, *8-11*, *8-13*, *8-20*
OCI_ATTR_UB8_ROW_COUNT, *6-19*
statement handle attribute, *A-41*
OCI_ATTR_USER_MEMORY, *12-5*
server handle attribute, *A-15*
OCI_ATTR_USERNAME
OCIAuthInfo handle attributes, *A-21*
user session handle attribute, *A-21*
OCI_ATTR_VALUE
attribute, *8-26*
OCI_ATTR_VARTYPE_MAXLEN_COMPAT
service context handle attribute, *A-10*
OCI_ATTR_VERSION
attribute, *8-23*
OCI_ATTR_VISIBILITY
OCIAQDeqOptions descriptor attribute, *A-77*
OCIAQEnqOptions descriptor attribute, *A-75*
OCI_ATTR_WAIT
OCIAQDeqOptions descriptor attribute, *A-77*
OCI_ATTR_WALL_LOC
environment handle attribute, *A-3*
OCI_ATTR_XID, *10-3*
transaction handle attribute, *A-40*
OCI_ATTR_XSTREAM_ACK_INTERVAL, *34-3*
OCI_ATTR_XSTREAM_IDLE_TIMEOUT, *34-4*
OCI_BIND_SOFT, *25-61*, *25-66*, *25-72*, *25-77*
OCI_CONTINUE, *4-27*
OCI_CPOOL_REINITIALIZE, *25-6*
OCI_CRED_EXT, *25-28*
OCI_CRED_PROXY, *10-16*
OCI_CRED_RDBMS, *10-16*, *25-28*
OCI_DATA_AT_EXEC, *25-61*, *25-66*, *25-72*, *25-77*
OCI_DEFAULT, *10-36*, *25-6*, *25-15*

OCI_DEFAULT mode for OCIDefineByPos(), 25-89
OCI_DEFAULT mode for OCIDefineByPos2(), 25-93
OCI_DEFINE_SOFT, 25-89, 25-93
OCI_DESCRIBE_ONLY, 26-2
OCI_DIRPATH_COL_ERROR, 21-21
OCI_DIRPATH_DATASAVE_FINISH, 26-104
OCI_DIRPATH_DATASAVE_SAVEONLY, 26-104
OCI_DIRPATH_EXPR_OBJ_CONSTR, 21-36
OCI_DIRPATH_EXPR_REF_TBLNAME, 21-25, 21-36
OCI_DIRPATH_EXPR_SQL, 21-36
OCI_DIRPATH_OID column array attribute, 21-41
OCI_DTYPE_AQAGENT, 4-12
OCI_DTYPE_AQDEQ_OPTIONS, 4-12
OCI_DTYPE_AQENQ_OPTIONS, 4-12
OCI_DTYPE_AQMSG_PROPERTIES, 4-12
OCI_DTYPE_AQNFY, 4-12
OCI_DTYPE_CHDES, 4-12
OCI_DTYPE_COMPLEXOBJECTCOMP, 4-12
OCI_DTYPE_DATE, 4-12
OCI_DTYPE_FILE, 4-12
OCI_DTYPE_INTERVAL_DS, 4-12
OCI_DTYPE_INTERVAL_YM, 4-12
OCI_DTYPE_LOB, 4-12
OCI_DTYPE_PARAM, 4-12, 25-48, 25-56
  when used, 25-48
OCI_DTYPE_ROWID, 4-12
OCI_DTYPE_SHARD_KEY, 4-13
OCI_DTYPE_SNAP, 4-12
OCI_DTYPE_SRVDN, 4-12
OCI_DTYPE_TIMESTAMP, 4-12
OCI_DTYPE_TIMESTAMP_LTZ, 4-12
OCI_DTYPE_TIMESTAMP_TZ, 4-12
OCI_DURATION_DEFAULT, 27-48
OCI_DURATION_SESSION, 22-7, 26-17, 29-7, 30-3, 30-13, 30-19, 30-26
OCI_DURATION_STATEMENT, 26-17, 29-7, 30-3, 30-13, 30-18, 30-26
OCI_DURATION_TRANS, 8-7, 22-7
  possible value of OCI_ATTR_DURATION, 8-7
OCI_DYNAMIC_FETCH, 25-89, 25-93
OCI_ENV_NO_MUTEX, 10-36
OCI_ERROR, 4-27, 10-7
OCI_EVENTS, 25-15
  mode for receiving notifications, 13-4
OCI_FOREIGN_SYNTAX, 26-10, E-7
OCI_HTYPE_ADMIN, 4-3, 16-1
OCI_HTYPE_AUTHINFO, 4-3, 11-6
OCI_HTYPE_BIND, 4-3
OCI_HTYPE_COMPLEXOBJECT, 4-3
OCI_HTYPE_COR, 25-56
OCI_HTYPE_CPOOL, 4-3, 11-18
OCI_HTYPE_DEFINE, 4-3
OCI_HTYPE_DESCRIBE, 4-3
OCI_HTYPE_DIRPATH_COLUMN_ARRAY, 4-3
OCI_HTYPE_DIRPATH_CTX, 4-3
OCI_HTYPE_DIRPATH_FN_COL_ARRAY, 21-20, 21-38
OCI_HTYPE_DIRPATH_FN_CTX, 4-3
OCI_HTYPE_DIRPATH_STREAM, 4-3
OCI_HTYPE_ENV, 4-3
OCI_HTYPE_ERROR, 4-3
OCI_HTYPE_PROC, 4-3
OCI_HTYPE_SERVER, 4-3
OCI_HTYPE_SESSION, 4-3
OCI_HTYPE_SODA_COLLECTION, 4-3
OCI_HTYPE_SODA_CURSOR, 4-3
OCI_HTYPE_SODA_DOCUMENT, 4-3
OCI_HTYPE_SPOOL, 4-3
OCI_HTYPE_STMT, 4-3, 25-48, 25-56
OCI_HTYPE_SUBSCRIPTION, 4-3
OCI_HTYPE_SVCCTX, 4-3
OCI_HTYPE_TRANS, 4-3
OCI_IND_NULL, 19-28
OCI_INVALID_HANDLE, 4-27
OCI_IOV, 7-25, 25-89, 25-93
OCI_LOCK_NONE, 22-13
OCI_LOCK_X, 22-13
OCI_LOCK_X_NOWAIT, 22-13
  parameter usage, 22-14
OCI_LOGON2_STMTCACHE, 15-2
OCI_LTYPE_ARG_FUNC list attribute, 8-22
OCI_LTYPE_ARG_PROC list attribute, 8-22
OCI_LTYPE_DB_SCH list attribute, 8-22
OCI_LTYPE_SCH_OBJ list attribute, 8-22
OCI_LTYPE_SUBPRG list attribute, 8-22
OCI_LTYPE_TYPE_ARG_FUNC list attribute, 8-22
OCI_LTYPE_TYPE_ARG_PROC list attribute, 8-22
OCI_LTYPE_TYPE_ATTR list attribute, 8-22
OCI_LTYPE_TYPE_METHOD list attribute, 8-22
OCI_MIGRATE
  OCISessionBegin() mode, 10-11, 25-28
OCI_NEED_DATA, 4-27
OCI_NEW_LENGTH_SEMANTICS, 25-11
OCI_NO_DATA, 4-27
OCI_NO_UCB, 25-15
OCI_OBJECT, 25-15
OCI_PIN_ANY, 22-7
OCI_PIN_LATEST, 22-7
OCI_PIN_RECENT, 22-7
OCI_PREP2_IMPL_RESULTS_CLIENT, E-7
OCI_PTYPE_ARG
  attributes, 8-20

OCI_PTYPE_ARG *(continued)*
  possible value of OCI_ATTR_PTYPE, *8-5*
OCI_PTYPE_COL
  attributes, *8-16*
OCI_PTYPE_COLL
  attributes, *8-13*
OCI_PTYPE_DATABASE
  attributes, *8-23*
OCI_PTYPE_EVALUATION CONTEXT
  attributes, *8-24*
OCI_PTYPE_FUNC
  attributes, *8-8*
  possible value of OCI_ATTR_PTYPE, *8-5*
OCI_PTYPE_LIST
  attributes, *8-22*
OCI_PTYPE_NAME_VALUE
  attributes, *8-26*
OCI_PTYPE_PKG
  attributes, *8-9*
OCI_PTYPE_PROC
  attributes, *8-8*
OCI_PTYPE_RULE_SET
  attributes, *8-24*
OCI_PTYPE_RULES
  attributes, *8-24*
OCI_PTYPE_SCHEMA
  attributes, *8-22*
OCI_PTYPE_SYN
  attributes, *8-14*
OCI_PTYPE_TABLE
  attributes, *8-7*
OCI_PTYPE_TABLE_ALIAS
  attributes, *8-25*
OCI_PTYPE_TYPE
  attributes, *8-9*
OCI_PTYPE_TYPE_ATTR
  attributes, *8-11*
OCI_PTYPE_TYPE_COLL
  possible value of OCI_ATTR_PTYPE, *8-5*
OCI_PTYPE_TYPE_FUNC
  attributes, *8-12*
OCI_PTYPE_TYPE_METHOD, *8-12*
OCI_PTYPE_TYPE_PROC
  attributes, *8-12*
OCI_PTYPE_UNK
  possible value of OCI_ATTR_PTYPE, *8-5*
OCI_PTYPE_VARIABLE_TYPE
  attributes, *8-25*
OCI_PTYPE_VIEW
  attributes, *8-7*
  possible value of OCI_ATTR_PTYPE, *8-5*
OCI_ROWCBK_DONE, *4-27*
OCI_SESSGET_CUSTOM_POOL mode, OCISessionGet(), *18-7*
OCI_SESSRLS_RETAG, *25-43*
OCI_STILL_EXECUTING, *4-27*
OCI_SUBSCR_NAMESPACE_ANONYMOUS, *A-86*
OCI_SUBSCR_PROTO_HTTP, *A-86*
OCI_SUBSCR_PROTO_MAIL, *A-86*
OCI_SUBSCR_PROTO_OCI, *A-86*
OCI_SUBSCR_PROTO_SERVER, *A-86*
OCI_SUBSCR_QOS_PURGE_ON_NTFN, *13-10*
OCI_SUBSCR_QOS_RELIABLE, *13-10*
OCI_SUCCESS, *4-27*, *10-7*
OCI_SUCCESS_WITH_INFO, *4-27*
OCI_SUPPRESS_NLS_VALIDATION, *25-11*, *25-15*
OCI_SYSASM
  OCISessionBegin() mode, *10-11*, *25-28*
OCI_SYSBKP
  OCISessionBegin() mode, *10-11*, *25-28*
OCI_SYSDBA
  OCISessionBegin() mode, *10-11*, *25-28*
OCI_SYSDGD
  OCISessionBegin() mode, *10-11*, *25-28*
OCI_SYSKMT
  OCISessionBegin() mode, *10-11*, *25-28*
OCI_SYSOPER
  OCISessionBegin() mode, *10-11*, *25-28*
OCI_SYSRAC
  OCISessionBegin() mode, *10-11*, *25-28*
OCI_THREADED, *10-36*, *25-15*
OCI_TRANS_LOOSE, *10-4*
OCI_TRANS_NEW, *10-9*
OCI_TRANS_READONLY, *10-10*
OCI_TRANS_RESUME, *10-9*
OCI_TRANS_SERIALIZABLE, *10-3*
OCI_TRANS_TWOPHASE, *10-9*
OCI_TYPECODE, *8-3*
  values, *5-35*, *5-36*
OCI_TYPECODE_ITABLE
  possible value of OCI_ATTR_COLLECTION_TYPECODE, *8-9*
OCI_TYPECODE_NCHAR, *20-32*
OCI_UTF16ID, *4-40*
oci.lib, *D-3*
OCIAnyDataAccess(), *30-8*
OCIAnyDataAttrGet(), *30-9*
OCIAnyDataAttrSet(), *30-11*
OCIAnyDataBeginCreate(), *30-13*
OCIAnyDataCollAddElem(), *30-15*
OCIAnyDataCollGetElem(), *30-17*
OCIAnyDataConvert(), *30-18*
OCIAnyDataDestroy(), *30-20*
OCIAnyDataEndCreate(), *30-20*
OCIAnyDataGetCurrAttrNum(), *30-21*
OCIAnyDataGetType(), *30-22*
OCIAnyDataIsNull(), *30-22*

OCIAnyDataSetAddInstance(), *30-25*
OCIAnyDataSetBeginCreate(), *30-26*
OCIAnyDataSetDestroy(), *30-27*
OCIAnyDataSetEndCreate(), *30-27*
OCIAnyDataSetGetCount(), *30-28*
OCIAnyDataSetGetInstance(), *30-29*
OCIAnyDataSetGetType(), *30-29*
OCIAnyDataTypeCodeToSqlt(), *20-32*, *30-23*
OCIAppCtxClearAll(), *25-4*
OCIAppCtxSet(), *25-5*
OCIAQAgent
    descriptor attributes, *A-85*
OCIAQDeq(), *26-79*
OCIAQDeqArray(), *26-82*
OCIAQDeqOptions
    descriptor attributes, *A-77*
OCIAQEnq(), *26-84*
OCIAQEnqArray(), *26-86*
OCIAQEnqOptions
    descriptor attributes, *A-75*
OCIAQListen(), *E-27*
OCIAQListen2(), *26-88*
OCIAQMsgProperties
    descriptor attributes, *A-81*
OCIArray, *20-17*
    binding and defining, *20-17*, *20-39*
OCIArray manipulation
    code example, *20-20*
OCIArrayDescriptorAlloc(), *25-46*
OCIArrayDescriptorFree(), *25-48*
OCIAttrGet(), *17-3*, *25-48*
    used for describing, *6-13*
OCIAttrSet(), *25-50*
OCIAuthInfo definition, *11-6*
OCIBindArrayOfStruct(), *25-59*
OCIBindByName(), *25-61*
OCIBindByName2(), *25-66*
OCIBindByPos(), *25-72*
OCIBindByPos2(), *25-77*
OCIBindDynamic(), *25-82*
OCIBindObject(), *25-85*
OCIBinXmlCreateReposCtxFromConn(), *32-2*
OCIBinXmlCreateReposCtxFromCPool(), *32-3*
OCIBinXmlSetFormatPref(), *23-6*, *32-4*
OCIBinXmlSetReposCtxForConn(), *32-4*
OCIBreak(), *26-152*
    use of, *4-33*, *4-38*
OCICacheFlush(), *27-6*
OCICacheFree(), *27-42*
OCICacheRefresh(), *27-8*
OCICacheUnmark(), *27-13*
OCICacheUnpin(), *27-44*
OCICharSetConversionIsReplacementUsed(), *31-42*
OCICharSetToUnicode(), *31-42*
OCIClientVersion(), *26-153*
OCIColl, *20-17*
    binding and defining, *20-17*
OCICollAppend(), *28-4*
OCICollAssign(), *28-5*
OCICollAssignElem(), *28-6*
OCICollGetElem(), *28-7*
OCICollGetElemArray(), *28-10*
OCICollIsLocator(), *28-12*
OCICollMax(), *28-13*
OCICollSize(), *28-14*
OCICollTrim(), *28-15*
OCIComplexObject
    use of, *19-23*
OCIComplexObjectComp
    use of, *19-23*
OCIConnectionPoolCreate(), *25-6*
OCIConnectionPoolDestroy(), *25-8*
OCIContextClearValue(), *29-11*
OCIContextGenerateKey(), *29-12*
OCIContextGetValue(), *29-13*
OCIContextSetValue(), *29-13*
OCIDate, *20-5*
    binding and defining, *20-5*, *20-39*
    manipulation usage example, *20-6*
OCIDateAddDays(), *28-27*
OCIDateAddMonths(), *28-28*
OCIDateAssign(), *28-29*
OCIDateCheck(), *28-29*
OCIDateCompare(), *28-31*
OCIDateDaysBetween(), *28-31*
OCIDateFromText(), *28-32*
OCIDateGetDate(), *28-33*
OCIDateGetTime(), *28-34*
OCIDateLastDay(), *28-35*
OCIDateNextDay(), *28-36*
OCIDateSetDate(), *28-37*
OCIDateSetTime(), *28-37*
OCIDateSysDate(), *28-38*
OCIDateTimeAssign(), *28-39*
OCIDateTimeCheck(), *28-40*
OCIDateTimeCompare(), *28-41*
OCIDateTimeConstruct(), *28-42*
OCIDateTimeConvert(), *28-44*
OCIDateTimeFromArray(), *28-44*
OCIDateTimeFromText(), *28-46*
OCIDateTimeGetDate(), *28-47*
OCIDateTimeGetTime(), *28-48*
OCIDateTimeGetTimeZoneName(), *28-49*
OCIDateTimeGetTimeZoneOffset(), *28-50*
OCIDateTimeIntervalAdd(), *28-51*
OCIDateTimeIntervalSub(), *28-52*
OCIDateTimeSubtract(), *28-53*
OCIDateTimeSysTimeStamp(), *28-54*
OCIDateTimeToArray(), *28-54*

OCIDateTimeToText(), *28-55*
OCIDateToText(), *28-57*
OCIDateZoneToZone(), *28-58*
OCIDBShutdown(), *25-9*
OCIDBStartup(), *25-10*
OCIDefineArrayOfStruct(), *25-88*
OCIDefineByPos(), *25-89*
OCIDefineByPos2(), *25-93*
OCIDefineDynamic(), *25-98*
OCIDefineObject(), *25-101*
OCIDescribeAny(), *25-103*
  limitations, *8-3*
  overview, *8-1*
  usage examples, *8-28*
OCIDescriptorAlloc(), *25-51*
OCIDescriptorFree(), *25-53*
OCIDirPathAbort(), *26-97*
OCIDirPathColArrayEntryGet(), *26-98*
OCIDirPathColArrayEntrySet(), *26-99*
OCIDirPathColArrayReset(), *26-101*
OCIDirPathColArrayRowGet(), *26-101*
OCIDirPathColArrayToStream(), *26-102*
OCIDirPathDataSave(), *26-104*
OCIDirPathFinish(), *26-105*
OCIDirPathFlushRow(), *26-106*
OCIDirPathLoadStream(), *26-107*
OCIDirPathPrepare(), *26-108*
OCIDirPathStreamReset(), *26-109*
OCIDuration
  use of, *22-7*
OCIDurationBegin(), *26-17*, *29-7*
OCIDurationEnd(), *26-18*, *29-8*
OCIEnvCreate(), *25-11*
OCIEnvInit(), *E-2*
OCIEnvNlsCreate(), *4-40*, *25-15*
OCIErrorGet(), *26-155*
OCIEvent handle, *12-3*
OCIExtProcAllocCallMemory(), *29-2*
OCIExtProcGetEnv(), *29-3*
OCIExtProcRaiseExcp(), *29-4*
OCIExtProcRaiseExcpWithMsg(), *29-5*
OCIExtractFromFile(), *29-15*
OCIExtractFromList(), *29-16*
OCIExtractFromStr(), *29-17*
OCIExtractInit(), *29-18*
OCIExtractReset(), *29-19*
OCIExtractSetKey(), *29-20*
OCIExtractSetNumKeys(), *29-21*
OCIExtractTerm(), *29-22*
OCIExtractToBool(), *29-22*
OCIExtractToInt(), *29-23*
OCIExtractToList(), *29-24*
OCIExtractToOCINum(), *29-25*
OCIExtractToStr(), *29-25*
OCIFileClose(), *29-27*
OCIFileExists(), *29-28*
OCIFileFlush(), *29-29*
OCIFileGetLength(), *29-29*
OCIFileInit(), *29-30*
OCIFileObject data structure, *29-26*
OCIFileOpen(), *29-31*
OCIFileRead(), *29-32*
OCIFileSeek(), *29-33*
OCIFileTerm(), *29-34*
OCIFileWrite(), *29-34*
OCIFormatInit(), *29-36*
OCIFormatString(), *29-36*
OCIFormatTerm(), *29-41*
OCIHandleAlloc(), *25-54*
OCIHandleFree(), *25-55*
OCIInd
  use of, *19-28*
OCIInitialize(), *E-3*
OCIIntervalAdd(), *28-59*
OCIIntervalAssign(), *28-60*
OCIIntervalCheck(), *28-61*
OCIIntervalCompare(), *28-62*
OCIIntervalDivide(), *28-63*
OCIIntervalFromNumber(), *28-64*
OCIIntervalFromText(), *28-65*
OCIIntervalFromTZ(), *28-66*
OCIIntervalGetDaySecond(), *28-67*
OCIIntervalGetYearMonth(), *28-68*
OCIIntervalMultiply(), *28-69*
OCIIntervalSetDaySecond(), *28-69*
OCIIntervalSetYearMonth(), *28-70*
OCIIntervalSubtract(), *28-71*
OCIIntervalToNumber(), *28-72*
OCIIntervalToText(), *28-73*
OCIIOV struct, *7-25*
OCIIter, *20-17*
  binding and defining, *20-17*
  usage example, *20-20*
OCIIterCreate(), *28-16*
OCIIterDelete(), *28-18*
OCIIterGetCurrent(), *28-19*
OCIIterInit(), *28-20*
OCIIterNext(), *28-22*
OCIIterPrev(), *28-23*
OCILCRAttributesGet(), *35-7*
OCILCRAttributesSet(), *35-9*
OCILCRComparePosition(), *35-10*
OCILCRConvertPosition(), *35-11*
OCILCRDDLInfoGet(), *35-13*
OCILCRDDLInfoSet(), *35-26*
OCILCRFree(), *35-12*
OCILCRGetLCRIDVersion(), *35-29*
OCILCRHeaderGet(), *35-15*
OCILCRHeaderSet(), *35-29*
OCILCRLobInfoGet(), *35-32*

OCILCRLobInfoSet(), *35-33*
OCILCRNew(), *35-20*
OCILCRNumberFromPosition(), *35-34*
OCILCRRowColumnInfoGet(), *35-21*
OCILCRRowColumnInfoSet(), *35-24*
OCILCRRowStmtGet(), *35-17*
OCILCRRowStmtWithBindVarGet(), *35-18*
OCILCRSCNToPosition(), *35-35*
OCILCRScnToPosition2(), *35-36*
OCILCRWhereClauseGet(), *35-37*
OCILCRWhereClauseWithBindVarGet(), *35-38*
OCILdaToSvcCtx(), *26-157*
OCILobAppend(), *26-19*
OCILobArrayRead(), *9-25*, *26-20*
OCILobArrayWrite(), *26-24*
OCILobAssign(), *26-28*
OCILobCharSetForm(), *26-30*
OCILobCharSetId(), *26-31*
OCILobClose(), *26-32*
OCILobCopy(), *E-9*
OCILobCopy2(), *26-33*
OCILobCreateTemporary(), *26-35*
OCILobDisableBuffering(), *E-10*
OCILobEnableBuffering(), *E-11*
OCILobErase(), *E-12*
OCILobErase2(), *26-37*
OCILobFileClose(), *26-38*
OCILobFileCloseAll(), *26-40*
OCILobFileExists(), *26-40*
OCILobFileGetName(), *26-42*
OCILobFileIsOpen(), *26-43*
OCILobFileOpen(), *26-44*
OCILobFileSetName(), *26-46*
OCILobFlushBuffer(), *E-13*
OCILobFreeTemporary(), *26-47*
OCILobGetChunkSize(), *26-48*
OCILobGetContentType(), *26-49*
OCILobGetLength(), *E-14*
OCILobGetLength2(), *26-50*
OCILobGetOptions(), *26-52*
OCILobGetStorageLimit(), *26-53*
OCILobIsEqual(), *26-54*
OCILobIsOpen(), *26-55*
OCILobIsTemporary(), *26-56*
OCILobLoadFromFile(), *E-15*
OCILobLoadFromFile2(), *26-57*
OCILobLocatorAssign(), *26-59*
OCILobLocatorIsInit(), *26-60*
OCILobOpen(), *26-62*
OCILobRead(), *E-15*
OCILobRead2(), *26-63*
OCILobSetContentType(), *26-67*
OCILobSetOptions(), *26-69*
OCILobTrim(), *E-19*
OCILobTrim2(), *26-70*
OCILobWrite(), *E-20*
OCILobWrite2(), *26-71*
OCILobWriteAppend(), *E-24*
OCILobWriteAppend2(), *26-75*
OCILockOpt
    possible values, *27-21*, *27-48*
OCILogoff(), *25-19*
OCILogon(), *25-20*
    using, *4-20*
OCILogon2(), *25-22*
OCIMemoryAlloc(), *29-8*
OCIMemoryFree(), *29-9*
OCIMemoryResize(), *29-10*
OCIMessageClose(), *31-46*
OCIMessageGet(), *31-47*
OCIMessageOpen(), *31-48*
OCIMultiByteInSizeToWideChar(), *31-12*
OCIMultiByteStrCaseConversion(), *31-13*
OCIMultiByteStrcat(), *31-14*
OCIMultiByteStrcmp(), *31-15*
OCIMultiByteStrcpy(), *31-16*
OCIMultiByteStrlen(), *31-16*
OCIMultiByteStrncat(), *31-17*
OCIMultiByteStrncmp(), *31-18*
OCIMultiByteStrncpy(), *31-19*
OCIMultiByteStrnDisplayLength(), *31-20*
OCIMultiByteToWideChar(), *31-21*
OCINlsCharSetConvert(), *31-43*
OCINlsCharSetIdToName(), *31-2*
OCINlsCharSetNameTold(), *31-3*
OCINlsEnvironmentVariableGet(), *31-4*
OCINlsGetInfo(), *31-5*
OCINlsNameMap(), *31-9*
OCINlsNumericInfoGet(), *31-7*
OCINumber, *20-12*
    bind example, *20-41*
    binding and defining, *20-12*, *20-39*
    define example, *20-41*
    usage examples, *20-12*
OCINumberAbs(), *28-76*
OCINumberAdd(), *28-77*
OCINumberArcCos(), *28-77*
OCINumberArcSin(), *28-78*
OCINumberArcTan(), *28-79*
OCINumberArcTan2(), *28-79*
OCINumberAssign(), *28-80*
OCINumberCeil(), *28-81*
OCINumberCmp(), *28-82*
OCINumberCos(), *28-82*
OCINumberDec(), *28-83*
OCINumberDiv(), *28-84*
OCINumberExp(), *28-85*
OCINumberFloor(), *28-85*
OCINumberFromInt(), *28-86*
OCINumberFromReal(), *28-87*

OCINumberFromText(), *28-88*
OCINumberHypCos(), *28-89*
OCINumberHypSin(), *28-90*
OCINumberHypTan(), *28-90*
OCINumberInc(), *28-91*
OCINumberIntPower(), *28-92*
OCINumberIsInt(), *28-93*
OCINumberIsZero(), *28-94*
OCINumberLn(), *28-94*
OCINumberLog(), *28-95*
OCINumberMod(), *28-96*
OCINumberMul(), *28-97*
OCINumberNeg(), *28-97*
OCINumberPower(), *28-98*
OCINumberPrec(), *28-99*
OCINumberRound(), *28-100*
OCINumberSetPi(), *28-100*
OCINumberSetZero(), *28-101*
OCINumberShift(), *28-102*
OCINumberSign(), *28-102*
OCINumberSin(), *28-103*
OCINumberSqrt(), *28-104*
OCINumberSub(), *28-105*
OCINumberTan(), *28-105*
OCINumberToInt(), *28-106*
OCINumberToReal(), *28-107*
OCINumberToRealArray(), *28-108*
OCINumberToText(), *28-109*
OCINumberTrunc(), *28-110*
OCIObjectArrayPin(), *27-45*
OCIObjectCopy(), *27-27*
OCIObjectExists(), *27-20*
OCIObjectFlush(), *27-10*
OCIObjectFree(), *27-46*
OCIObjectGetAttr(), *27-29*
OCIObjectGetInd(), *27-31*
    example of use, *19-28*
OCIObjectGetObjectRef(), *27-32*
OCIObjectGetProperty(), *27-21*
OCIObjectGetTypeRef(), *27-33*
OCIObjectIsDirty(), *27-24*
OCIObjectIsLocked(), *27-25*
OCIObjectLifetime
    possible values, *27-21*
OCIObjectLock(), *27-34*
OCIObjectLockNoWait(), *27-35*
OCIObjectMarkDelete(), *27-14*
OCIObjectMarkDeleteByRef(), *27-15*
OCIObjectMarkStatus
    possible values, *27-21*
OCIObjectMarkUpdate(), *27-16*
OCIObjectNew(), *27-36*
OCIObjectPin(), *27-48*
OCIObjectPinCountReset(), *27-50*
OCIObjectPinTable(), *27-51*
OCIObjectRefresh(), *27-11*
OCIObjectSetAttr(), *27-40*
OCIObjectUnmark(), *27-18*
OCIObjectUnmarkByRef(), *27-19*
OCIObjectUnpin(), *27-53*
OCIParamGet(), *25-56*
    used for describing, *6-13*
OCIParamSet(), *25-57*
OCIPasswordChange(), *26-158*
OCIPing(), *26-160*
OCIPinOpt
    use of, *22-7*
OCIRaw, *20-16*
    binding and defining, *20-16*, *20-39*
    manipulation usage example, *20-17*
OCIRawAllocSize(), *28-111*
OCIRawAssignBytes(), *28-112*
OCIRawAssignRaw(), *28-113*
OCIRawPtr(), *28-114*
OCIRawResize(), *28-115*
OCIRawSize(), *28-116*
OCIRef, *20-24*
    binding and defining, *20-24*
    usage example, *20-25*
OCIRefAssign(), *28-117*
OCIRefClear(), *28-118*
OCIRefFromHex(), *28-119*
OCIRefHexSize(), *28-120*
OCIRefIsEqual(), *28-121*
OCIRefIsNull(), *28-122*
OCIRefToHex(), *28-123*
OCIRequestEnd(), *25-24*
OCIReset(), *26-161*
    use of, *4-38*
OCIRowid ROWID descriptor, *4-15*
OCIRowidToChar(), *26-162*
OCIServerAttach(), *25-25*
    shadow processes, *25-25*
OCIServerDetach(), *25-27*
OCIServerDNs descriptor attributes, *A-85*
OCIServerRelease(), *26-163*
OCIServerRelease2(), *26-164*
OCIServerVersion(), *26-166*
OCISessionBegin(), *10-11*, *25-28*
OCISessionEnd(), *25-31*
OCISessionGet(), *25-32*
OCISessionPoolCreate(), *25-39*
OCISessionPoolDestroy(), *25-43*
OCISessionRelease(), *25-43*
OCIShardingKeyColumnAdd(), *26-146*
OCIShardingkeyReset(), *26-148*
OCIShardingKeyReset(), *18-1*
OCIShardInstancesGet(), *18-7*, *26-149*
OCISodaBulkInsert(), *36-3*
OCISodaBulkInsertAndGet(), *36-4*

OCISodaBulkInsertAndGetWithCont(), *36-6*
OCISodaBulkInsertWithCtnt(), *36-9*
OCISodaCollCreate(), *36-11*
OCISodaCollCreateWithMetadata(), *36-13*
OCISodaCollDrop(), *36-14*
OCISodaCollList(), *36-17*
OCISodaCollOpen(), *36-18*
OCISodaDataGuideGet(), *36-20*
OCISodaDocCount(), *36-22*
OCISodaDocCountWithFilter(), *36-24*
OCISodaDocCreate(), *36-25*
OCISodaDocCreateWithKey(), *36-27*
OCISodaDocCreateWithKeyAndMType(), *36-28*
OCISodaDocGetNext(), *36-30*
OCISodaFind(), *36-31*
OCISodaFindOne(), *36-34*
OCISodaFindOneWithKey(), *36-35*
OCISodaIndexCreate(), *36-37*
OCISodaIndexDrop(), *36-39*
OCISodaInsert(), *36-40*
OCISodaInsertAndGet(), *36-41*
OCISodaInsertAndGetWithCtnt(), *36-43*
OCISodaInsertWithCtnt(), *36-45*
OCISodaRemove(), *36-48*
OCISodaRemoveOneWithKey(), *36-49*
OCISodaReplOne(), *36-50*
OCISodaReplOneAndGet(), *36-52*
OCISodaReplOneAndGetWithCtnt(), *36-54*
OCISodaReplOneAndGetWithKey(), *36-57*
OCISodaReplOneWithCont(), *36-59*
OCISodaReplOneWithKey(), *36-62*
OCIStmtExecute(), *26-2*
    prefetch during, *6-7*
    use of iters parameter, *6-7*
OCIStmtFetch(), *17-3*, *E-6*
OCIStmtFetch2(), *17-3*, *26-5*
OCIStmtGetBindInfo(), *25-106*
OCIStmtGetNextResult (), *26-7*
OCIStmtGetPieceInfo(), *26-9*
OCIStmtPrepare(), *E-7*
OCIStmtPrepare2(), *26-10*
    preparing SQL statements, *6-4*
OCIStmtRelease(), *26-12*
OCIStmtSetPieceInfo(), *26-13*
OCIString, *20-14*
    binding and defining, *20-14*, *20-39*
    manipulation usage example, *20-15*
OCIStringAllocSize(), *28-125*
OCIStringAssign(), *28-126*
OCIStringAssignText(), *28-127*
OCIStringPtr(), *28-128*
OCIStringResize(), *28-129*
OCIStringSize(), *28-130*
OCISubscriptionDisable(), *26-89*
OCISubscriptionEnable(), *26-90*
OCISubscriptionPost(), *26-92*
OCISubscriptionRegister(), *17-3*, *26-93*
OCISubscriptionUnRegister(), *17-3*, *26-95*
OCISvcCtxToLda(), *26-167*
OCITable, *20-17*
    binding and defining, *20-17*, *20-39*
OCITableDelete(), *28-131*
OCITableExists(), *28-132*
OCITableFirst(), *28-133*
OCITableLast(), *28-135*
OCITableNext(), *28-136*
OCITablePrev(), *28-137*
OCITableSize(), *28-138*
OCITerminate(), *25-45*
OCIThread package, *10-37*
OCIThreadClose(), *26-111*
OCIThreadCreate(), *26-112*
OCIThreadHandleGet(), *26-113*
OCIThreadHndDestroy(), *26-114*
OCIThreadHndInit(), *26-114*
OCIThreadIdDestroy(), *26-115*
OCIThreadIdGet(), *26-116*
OCIThreadIdInit(), *26-117*
OCIThreadIdNull(), *26-118*
OCIThreadIdSame(), *26-119*
OCIThreadIdSet(), *26-120*
OCIThreadIdSetNull(), *26-121*
OCIThreadInit(), *26-122*
OCIThreadIsMulti(), *26-122*
OCIThreadJoin(), *26-123*
OCIThreadKeyDestroy(), *26-124*
OCIThreadKeyGet(), *26-125*
OCIThreadKeyInit(), *26-126*
OCIThreadKeySet(), *26-127*
OCIThreadMutexAcquire(), *26-128*
OCIThreadMutexDestroy(), *26-128*
OCIThreadMutexInit(), *26-129*
OCIThreadMutexRelease(), *26-130*
OCIThreadProcessInit(), *26-131*
OCIThreadTerm(), *26-132*
OCITransCommit(), *17-3*, *26-133*
OCITransDetach(), *17-3*, *26-135*
OCITransForget(), *17-3*, *26-136*
OCITransMultiPrepare(), *17-3*, *26-137*
OCITransPrepare(), *17-3*, *26-138*
OCITransRollback(), *17-3*, *26-139*
OCITransStart(), *17-3*, *26-140*
OCIType
    description, *20-25*
OCITypeAddAttr(), *30-2*
OCITypeArrayByFullName(), *27-58*
OCITypeArrayByName(), *27-55*
OCITypeArrayByRef(), *27-60*
OCITypeBeginCreate(), *30-3*
OCITypeByName(), *27-65*

OCITypeByRef(), *27-67*
OCITypeCode, *5-35*
OCITypeElem
    description, *20-25*
OCITypeEndCreate(), *30-5*
OCITypeMethod
    description, *20-25*
OCITypePackage(), *27-69*
OCITypeSetBuiltin(), *30-5*
OCITypeSetCollection(), *30-6*
OCIUnicodeToCharSet(), *31-45*
OCIUserCallbackGet(), *26-168*
OCIUserCallbackRegister(), *26-169*
OCIWchar data type, *4-44*
OCIWideCharInSizeToMultiByte(), *31-22*
OCIWideCharIsAlnum(), *31-35*
OCIWideCharIsAlpha(), *31-36*
OCIWideCharIsCntrl(), *31-36*
OCIWideCharIsDigit(), *31-37*
OCIWideCharIsGraph(), *31-37*
OCIWideCharIsLower(), *31-38*
OCIWideCharIsPrint(), *31-38*
OCIWideCharIsPunct(), *31-39*
OCIWideCharIsSingleByte(), *31-39*
OCIWideCharIsSpace(), *31-40*
OCIWideCharIsUpper(), *31-40*
OCIWideCharIsXdigit(), *31-41*
OCIWideCharMultiByteLength(), *31-23*
OCIWideCharStrCaseConversion(), *31-23*
OCIWideCharStrcat(), *31-24*
OCIWideCharStrchr(), *31-25*
OCIWideCharStrcmp(), *31-26*
OCIWideCharStrcpy(), *31-27*
OCIWideCharStrlen(), *31-28*
OCIWideCharStrncat(), *31-28*
OCIWideCharStrncmp(), *31-29*
OCIWideCharStrncpy(), *31-30*
OCIWideCharStrrchr(), *31-31*
OCIWideCharToLower(), *31-32*
OCIWideCharToMultiByte(), *31-33*
OCIWideCharToUpper(), *31-34*
ocixmldb.h header file, *23-2*
OCIXmlDbFreeXmlCtx(), *32-5*
OCIXmlDbInitXmlCtx(), *32-5*
OCIXStreamInAttach(), *35-40*
OCIXStreamInChunkSend(), *35-51*
OCIXStreamInCommit(), *35-54*
OCIXStreamInDetach(), *35-41*
OCIXStreamInErrorGet(), *35-49*
OCIXStreamInFlush(), *35-50*
OCIXStreamInLCRCallbackSend(), *35-43*
OCIXStreamInLCRSend(), *35-42*
OCIXStreamInProcessedLWMGet(), *35-48*
OCIXStreamInSessionSet(), *35-55*
OCIXStreamOutAttach(), *35-56*
OCIXStreamOutChunkReceive(), *35-66*
OCIXStreamOutDetach(), *35-58*
OCIXStreamOutGetNextChunk(), *35-69*
OCIXStreamOutLCRCallbackReceive(), *35-60*
OCIXStreamOutLCRReceive(), *35-58*
OCIXStreamOutProcessedLWMSet(), *35-65*
OCIXStreamOutSessionSet(), *35-72*
ODPI-C
    alternative to OCI, *1-3*
    for Oracle Database drivers, *1-3*
    for user applications, *1-3*
OID
    *See* object identifiers
opaque, definition of, *1-1*
optimistic locking
    implementing, *22-14*
ORA_NCHAR_LITERAL_REPLACE, *25-11*
ORA-25219 error during enqueue, *13-25*
Oracle Call Interface
    *See* OCI
Oracle Connection Manager in Traffic Director Mode, *11-12*
Oracle data types, *5-1*
    mapping to C, *20-2*
Oracle Database, *D-4*
Oracle Database Advanced Queuing
    dequeue function, *26-79*
    description, *13-22*
    descriptor attributes, *A-75*
    enqueue function, *26-84*
    functions, *26-79*, *E-27*
    OCI and, *13-22*
    OCI versus PL/SQL, *13-25*
    publish-subscribe notification in OCI, *13-2*
Oracle directory structure, *3-2*
Oracle ODBC Driver
    for user ODBC applications, *1-3*
Oracle RAC, *12-1*
Oracle Real Application Clusters, *12-1*
Oracle Streams Advanced Queuing
    OCI descriptors for, *13-24*
    OCI functions for, *13-23*
Oracle XA Library
    additional documentation, *D-6*
    compiling and linking an OCI program, *D-5*
    dynamic registration, *D-5*
Oracle XML DB OCI functions, *23-1*, *32-2*
oratypes.h
    as parameter source in OCI, *5-38*
    contents, *5-38*
    definitions in, *5-38*
ORE
    *See* object runtime environment
OTT
    *See* OTT utility

OTT parameters, 24-24, 24-25
  CASE, 24-29
  CODE, 24-27
  CONFIG, 24-28
  ERRTYPE, 24-28
  HFILE, 24-28
  INITFILE, 24-27
  INITFUNC, 24-28
  INTYPE, 24-26
  OUTTYPE, 24-27
  SCHEMA_NAMES, 24-29
  TRANSITIVE, 24-30
  URL, 24-30
  USERID, 24-26
  where they appear, 24-30
OTT utility, 19-8
  command line, 24-5, D-6
  command-line syntax, 24-24
  creating types in the database, 24-3
  data type mappings, 24-9
  initialization function
    calling, 24-21
    tasks of, 24-23
  intype file, 24-31
  outtype file, 24-18
  overview, 24-1
  parameters, 24-25
    See OTT parameters, 24-25
  providing an intype file, 24-8
  reference, 24-23
  restriction, 24-37
  sample output, 19-8
  use with OCI, 19-8
  using, 24-1
outbound servers
  OCI interface, 34-1
outtype file, 24-31
  when running OTT utility, 24-18
OUTTYPE OTT parameter, 24-27

P

packages
  attributes, 8-9
  describing, 8-1
parameter descriptor, 4-15
  attributes, 8-5, A-58
  object, 20-25
parameters
  attributes, 8-5
  buffer lengths, 25-3
  modes, 25-2
  passing strings, 4-31
  string length, 25-2
password management, 10-11, 10-13
persistent objects, 19-4
  meta-attributes, 19-17
piecewise
  binds and defines for LOBs, 7-54
  fetch, 7-52
piecewise operations, 7-47
  fetch, 7-45, 7-52
  in PL/SQL, 7-50
  insert, 7-45
  update, 7-45
  valid data types, 7-46
pin count, 19-28
pin duration
  example, 22-16
  of objects, 22-15
pinning objects, 22-7
PL/SQL, 1-7
  binding and defining nested tables, 7-43
  binding and defining REF CURSORS, 7-43
  binding placeholders, 4-38
  defining output variables, 7-22
  piecewise operations, 7-50
  uses in OCI applications, 4-38
  using in OCI applications, 4-38
  using in OCI programs, 7-7
PL/SQL Callback
  for session state fix up, 11-4
placeholders
  rules, 6-7
pluggable databases
  OCI support for, 17-1
polling mode, 4-36
positioned deletes, 4-34
positioned updates, 4-34
prefetching
  during OCIStmtExecute(), 6-7
  LOBs, 9-25
  OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE, 9-25
  setting prefetch memory size, 6-17
  setting row count, 6-17
Pro*C/C++
  alternative to OCI, 1-3
procedures
  attributes, 8-8
process
  handle attributes, A-114
proxy access, 4-21
proxy authentication, 4-21, 10-19
publish-subscribe
  _SYSTEM_TRIG_ENABLED parameter, 13-15
  COMPATIBLE parameter, 13-4
  example, 13-15
  functions, 13-4, 26-79, E-27

publish-subscribe *(continued)*
  handle attributes, *13-4*, *A-86*
  LDAP registration, *13-9*
  notification callback, *13-12*
  notification in OCI, *13-2*
  subscription handle, *13-4*

## Q

query, *6-16*
  explicit describe, *6-15*
    *See also* SQL query

## R

ram_threshold, *15-15*
RAW
  external data type, *5-18*
read-only parameter descriptor, *4-12*
REF
  external data type, *5-22*
REF columns
  direct path loading of, *21-24*
REF CURSOR variables
  binding and defining, *7-43*
references to objects
  *See* REFs
refreshing, *22-12*
  objects, *22-12*
REFs, *19-10*
  binding, *20-34*
  CURSOR variables, binding, *7-17*
  defining, *20-37*
  external data type, *5-22*
  indicator variables for, *4-31*, *4-32*
  retrieving from server, *19-10*
registering
  user callbacks, *14-1*
registry
  REGEDT32, *D-6*
relational functions, *C-7*
  server round-trips, *C-2*
relinking, need for, *3-20*
required support files, *D-1*
reserved namespaces, *4-35*
reserved words, *4-35*
restrictions
  on OCI API calls with CDBs in general, *17-2*
  on OCI calls with ALTER SESSION SET CONTAINER, *17-3*
  on OCI calls with ALTER SESSION SWITCH CONTAINER SWITCH SERVICE, *17-5*
result cache, *15-12*
result set, *6-19*
result set descriptor, *4-12*
resuming branches, *10-6*
retrieving attributes of an object type
  example, *8-31*
retrieving column data types for a table
  example, *8-28*
retrieving the collection element's data type of a named collection type
  example, *8-33*
return values
  navigational functions, *27-4*
RETURNING clause
  binding with, *7-30*
  error handling, *7-31*
  initializing variables, *7-30*
  using with OCI, *7-29*
  with REFs, *7-31*
rollback, *4-25*
  in object applications, *22-15*
row change descriptor, *4-13*
ROWID
  external data type, *5-23*
  implicit fetching, *15-6*
  logical, *5-6*
  OCIRowid descriptor, *4-15*
  Universal ROWID, *5-6*
  used for positioned updates and deletes, *4-34*
ROWID descriptor, *4-12*
RSFs, *D-1*
rule sets
  attributes, *8-24*
    type OCI_PTYPE_RULE_SET, *8-24*
rules
  attributes, *8-24*
    type OCI_PTYPE_RULE, *8-24*
running OCI application, *D-4*

## S

sample programs, *B-1*, *D-2*
samples directory, *D-2*
sb1
  definition, *5-38*
sb2
  definition, *5-38*
sb4
  definition, *5-38*
scatter/gather for binds/defines, *7-25*
schema type attributes
  type OCI_PTYPE_SCHEMA, *8-22*
SCHEMA_NAMES OTT parameter, *24-29*
  usage, *24-34*
schemas
  attributes, *8-22*, *8-24*–*8-26*

schemas *(continued)*
  describing, *8-1*
scripts
  authenticating users in, *10-13*
scrollable cursor
  example, *6-21*
  in OCI, *6-19*
  increasing performance, *6-20*
SDK for Instant Client Light, *2-7*
secondary memory
  of object, *22-17*
SECUREFILE parameter, *9-28*
SecureFiles, *9-28*
  OCILobGetContentType(), *26-49*
  OCILobSetContentType(), *26-67*
SecureFiles LOBs, *9-28*
security
  init.ora parameters, *10-32*
  OCI enhancements, *10-32*
select list
  describing, *6-12*
  implicit describe, *6-12*
sequences
  attributes, *8-15*
  describing, *8-1*
server handle
  attributes, *A-15*
  description, *4-6*
  setting in service context, *4-6*
server round-trips
  cache functions, *C-5*
  data type mapping and manipulation functions, *C-7*
  definition of, *C-1*
  describe operation, *C-6*
  LOB functions, *C-3*
  object functions, *C-5*
  relational functions, *C-7*
service context handle
  attributes, *A-10*
  description, *4-6*
  elements of, *4-6*
session creation, *4-19*
session management, *10-11*, *10-14*
session migration, *10-11*, *25-28*
session pool handle
  attributes, *A-36*
session pooling, *11-1*, *11-23*
  example, *11-11*
  functionality, *11-2*
  runtime connection load balancing, *12-1*
  tagging, *11-1*
shard, *18-1*
Shard Instance Descriptor
  attributes, *A-59*
shard key or group key descriptor, *4-13*
sharding functions, *26-145*
sharding key, *18-1*
shutting down databases, *16-1*
signal handler, *10-34*
skip parameters
  for arrays of structures, *7-23*
  for standard arrays, *7-24*
snapshot descriptor, *4-12*, *4-14*
snapshots
  executing against, *6-8*
SODA collection handle attributes, *A-62*
SODA document handle attributes, *A-60*
SODA OCI client
  character sets, *36-1*
specifying a sharding key, *18-5*
specifying a super sharding key, *18-5*
SQL query, *6-16*
  binding placeholders
    See bind operations, *6-16*
  defining output variables, *6-16*, *7-17*, *20-36*
    See define operations, *6-16*
  fetching results, *6-16*
  statement type, *1-7*
SQL statements, *1-5*
  binding placeholders in, *6-6*, *7-1*, *20-33*
  determining type prepared, *6-4*
  executing, *6-7*
  preparing for execution, *6-4*
  processing, *6-1*
  types
    control statements, *1-6*
    data definition language, *1-5*
    data manipulation language, *1-6*
    embedded SQL, *1-8*
    PL/SQL, *1-7*
    queries, *1-7*
SQLCS_IMPLICIT, *7-35*, *26-20*, *26-24*, *26-30*, *26-35*, *26-71*, *26-75*, *E-15*, *E-20*, *E-24*
SQLCS_NCHAR, *26-24*, *26-30*, *26-35*, *26-63*, *26-71*, *E-20*, *E-24*
sqlnet.ora, controlling ADR, *3-29*
SQLT typecodes, *5-36*
SQLT_BDOUBLE, *5-29*
SQLT_BFLOAT, *5-29*
SQLT_CHR, *A-99*
SQLT_IBDOUBLE, *5-7*, *8-16*
SQLT_IBFLOAT, *5-7*, *8-16*
SQLT_NTY
  bind example, *20-46*
  define example, *20-47*
  description, *5-22*
  preallocating object memory, *20-37*
SQLT_REF
  definition, *5-22*

SQLT_REF *(continued)*
  description, *5-22*
starting up databases, *16-1*
stateful sessions, *11-16*
stateless sessions, *11-16*
statement caching, *15-1*
  code example, *15-6*
statement handle
  attributes, *A-41*
  description, *4-8*
statement_cache, *15-14*
statically linked applications, *3-20*
stored functions
  describing, *8-1*
stored procedures
  describing, *8-1*
STRING
  external data type, *5-15*
strings
  passing as parameters, *4-31*
structures
  arrays of, *7-22*
subprogram attributes, *8-8*
subscription handle, *4-10*
  attributes, *A-86*
super sharding key, *18-1*
supporting UTF-16 Unicode in OCI, *4-40*, *4-42*, *4-44*, *4-45*
sword
  definition in oratypes.h file, *5-38*
synonyms
  attributes, *8-14*
  describing, *8-1*

T

table alias
  attributes
    type OCI_PTYPE_TABLE_ALIAS, *8-25*
table change descriptor, *4-13*
tables
  attributes, *8-7*
  describing, *8-1*
  limitations on OCIDescribeAny() and OCIStmtExecute(), *8-3*
TAF (transparent application failover)
  callback registration, *12-11*
  callbacks, *12-7*
  connections enabled by, *12-6*
  in connection pools, *11-17*
  OCI callbacks, *12-7*
TAF failover callback
  structure and parameters, *12-9*
tagging
  custom pooling, *12-5*
tagging *(continued)*
  session pooling, *11-1*, *25-32*, *25-43*
task thread, *F-2*
TDO
  definition, *20-33*
  description, *20-25*
  in the object cache, *22-23*
  obtaining, *20-25*
terminology
  navigational functions, *27-3*
  used in this manual, *1-9*
thread
  dispatcher, *F-2*
  monitor, *F-2*
  task, *F-2*
thread handle description, *4-10*, *10-43*
thread management functions, *26-110*
thread safety, *10-34*
  advantages of, *10-35*
  basic concepts, *10-34*
  implementing with OCI, *10-36*
  mixing 7.x and 8.0 calls, *10-37*
  required OCI calls, *10-36*
  three-tier architectures, *10-35*
threads package, *10-37*
three-tier architectures
  thread safety, *10-35*
time zone files differ on client and server, *3-6*
TIMESTAMP data type, *5-27*
TIMESTAMP descriptor, *4-12*
TIMESTAMP WITH LOCAL TIME ZONE data type, *5-28*
TIMESTAMP WITH LOCAL TIME ZONE descriptor, *4-12*
TIMESTAMP WITH TIME ZONE data type, *5-27*
TIMESTAMP WITH TIME ZONE descriptor, *4-12*
TimesTen In-Memory Database access from OCI, *1-1*, *15-1*
TNS_ADMIN, *2-6*, *3-4*
tnsnames.ora file, *2-3*
top-level memory
  of object, *22-17*
Transaction Guard
  logical transaction ID (LTXID), *12-14*
transaction handle
  attributes, *A-40*
  description, *4-6*
transaction identifier, *10-3*
transaction processing monitor, *D-4*
  additional documentation, *D-6*
  types, *D-4*
transactional complexity
  levels in OCI, *10-2*
transactions
  committing, *4-25*

transactions *(continued)*
    functions, *26-132*
    global, *10-3*
        branch states, *10-6*
        branches, *10-4*
        one-phase commit, *10-7*
        transactions identifier, *10-3*
        two-phase commit, *10-7*
    global examples, *10-8*
    initialization parameters, *10-8*
    local, *10-2*
    OCI functions for
        transactions, *10-1*
    read-only, *10-3*
    rolling back, *4-25*
    serializable, *10-3*
transient objects, *19-5*
    LOBs
        attributes, *9-4*
    meta-attributes, *19-20*
TRANSITIVE OTT parameter, *24-8*, *24-12*, *24-30*
transparent application failover
    *See* TAF
type attributes
    attributes, *8-11*
type descriptor object, *19-8*, *20-25*
type evolution, *19-41*
    object cache, *22-23*
type functions
    attributes, *8-12*
type inheritance
    OTT utility support, *24-15*
type method attributes, *8-12*
type procedures
    attributes, *8-12*
type reference, *19-33*
typecodes, *5-35*
types
    attributes, *8-9*
    describing, *8-1*

## U

ub1
    definition, *5-38*
ub2
    definition, *5-38*
UB2MAXVAL, *25-77*
ub4
    definition, *5-38*
UB4MAXVAL, *6-19*
Unicode
    character set ID, *A-51*, *A-54*
    data buffer alignment, *4-45*
    OCILobRead(), *E-15*
Unicode *(continued)*
    OCILobWrite(), *26-24*, *26-71*, *E-20*
Universal ROWID, *5-6*
unmarking objects, *22-10*
unpinning objects, *19-28*, *22-8*
UNSIGNED
    external data type, *5-19*
updates
    piecewise, *7-45*, *7-47*
    positioned, *4-34*
upgrading OCI, *3-20*
URL OTT parameter, *24-30*
UROWID
    Universal ROWID, *5-6*
user callback descriptor, *4-12*
user memory
    allocating, *4-17*
user session handle
    attributes, *A-21*
    description, *4-6*
    setting in service context, *4-6*
user-defined callback functions, *14-1*
    registering, *14-1*
USERID OTT parameter, *24-26*
using an explicit describe on a named collection type
    example, *8-33*
using an explicit describe on a named object type
    example, *8-31*
using an explicit describe to retrieve column data types for a table
    example, *8-28*
utext
    Unicode data type, *7-36*, *7-42*
UTF-16 data, sample code, *7-42*

## V

validating format for
    Oracle NUMBER and DATE data, *21-16*
values, *19-4*
    in object applications, *19-6*
VARCHAR
    external data type, *5-17*
VARCHAR2
    external data type, *5-10*
variable type attributes
    type OCI_PTYPE_VARIABLE_TYPE, *8-25*
VARNUM
    external data type, *5-16*
VARRAW
    external data type, *5-19*
varrays, *5-22*
    as objects, *19-1*
    C mapping, *24-11*

varrays *(continued)*
  define calls, *20-36*
  iterator, *28-16*
  NULLs, *19-28*
  or collection Iterator example, *20-20*
  scan, *28-20*
version compatibility, *3-20*
views
  attributes, *8-7*
  describing, *8-1*

## W

wchar_t data type, *4-44*, *7-42*, *31-10*
with_context
  argument to external procedure functions, *29-1*

## X

XA Library
  compiling and linking an OCI program, *D-5*
  functions, *D-4*
  OCI support, *1-16*
  overview, *D-4*
XA specification, *10-3*
xa.h header, *1-16*
XID
  *See* transaction identifier
XML DB functions, *23-2*, *32-2*
XML support in OCI, *23-1*, *32-1*
XStream
  OCI interface, *34-1*
    character sets, *34-2*
    functions, *35-1*
    handler and descriptor attributes, *34-3*
    XStream In, *34-2*
    XStream Out, *34-1*
xtramem_sz parameter
  using, *4-17*