Oracle® Call Interface
Programmer's Guide
11g Release 2 (11.2)
E10646-15

July 2014
Preface ............................................................................................................................................................ xlvii
   Audience .................................................................................................................................................... xlvii
   Documentation Accessibility ....................................................................................................................... xlvii
   Related Documents ................................................................................................................................ xlviii
   Conventions ............................................................................................................................................ xlviii

What's New in Oracle Call Interface? ........................................................................................................ li
   New Features in Oracle Call Interface Release 11.2 ................................................................................ li
   New Features in Oracle Call Interface Release 11.1 ............................................................................ lii

1 OCI: Introduction and Upgrading

Overview of OCI ........................................................................................................................................... 1-1
   Advantages of OCI .................................................................................................................................. 1-2
   Building an OCI Application .................................................................................................................. 1-2
   Parts of OCI .......................................................................................................................................... 1-3
   Procedural and Nonprocedural Elements ............................................................................................... 1-3
   Object Support ...................................................................................................................................... 1-3
   SQL Statements ...................................................................................................................................... 1-4
     Data Definition Language ................................................................................................................... 1-5
     Control Statements .............................................................................................................................. 1-5
     Data Manipulation Language ............................................................................................................ 1-5
     Queries ................................................................................................................................................. 1-6
     PL/SQL ................................................................................................................................................. 1-6
     Embedded SQL .................................................................................................................................... 1-7
     Special OCI Terms for SQL ................................................................................................................ 1-7
   Encapsulated Interfaces ......................................................................................................................... 1-8
   Simplified User Authentication and Password Management .................................................................. 1-8
   Extensions to Improve Application Performance and Scalability ..................................................... 1-8
   OCI Object Support ............................................................................................................................... 1-9
     Client-Side Object Cache .................................................................................................................... 1-9
     Associative and Navigational Interfaces .............................................................................................. 1-10
     OCI Runtime Environment for Objects ................................................................................................ 1-10
     Type Management: Mapping and Manipulation Functions ................................................................ 1-10
     Object Type Translator ...................................................................................................................... 1-11
   OCI Support for Oracle Streams Advanced Queuing ........................................................................ 1-11
2 OCI Programming Basics

Header File and Makefile Locations ............................................................... 2-1
Overview of OCI Program Programming ..................................................... 2-2
OCI Data Structures .................................................................................... 2-3
Handles ......................................................................................................... 2-3
Allocating and Freeing Handles ................................................................. 2-4
Environment Handle .................................................................................. 2-5
Error Handle ............................................................................................... 2-5
Service Context Handle and Associated Handles ...................................... 2-5
Statement, Bind, and Define Handles .......................................................... 2-6
Describe Handle ........................................................................................ 2-7
Complex Object Retrieval Handle ............................................................. 2-7
Thread Handle ............................................................................................ 2-7
3 Data Types

Oracle Data Types ................................................................. 3-1
Using External Data Type Codes ........................................... 3-3
Internal Data Types .............................................................. 3-3
  LONG, RAW, LONG RAW, VARCHAR2 .................................... 3-4
  Character Strings and Byte Arrays ........................................ 3-4
  UROWID ............................................................................ 3-5
  BINARY_FLOAT and BINARY_DOUBLE ................................ 3-5
External Data Types ............................................................. 3-6
  VARCHAR2 ........................................................................ 3-7
    Input ............................................................................. 3-8
    Output .......................................................................... 3-8
  NUMBER .......................................................................... 3-8
  64-Bit Integer Host Data Type ............................................. 3-9
    OCI Bind and Define for 64-Bit Integers ........................... 3-10
    Support for OUT Bind DML Returning Statements ............ 3-10
  INTEGER ......................................................................... 3-11
  FLOAT ............................................................................. 3-11
  STRING ........................................................................... 3-11
    Input ............................................................................. 3-11
    Output .......................................................................... 3-12
  VARNUM .......................................................................... 3-12
  LONG .............................................................................. 3-12
  VARCHAR ......................................................................... 3-12
  DATE .............................................................................. 3-13
  RAW ............................................................................... 3-13
  VARRAW ......................................................................... 3-14
  LONG RAW ....................................................................... 3-14
  UNSIGNED ........................................................................ 3-14
  LONG VARCHAR .................................................................. 3-14
  LONG VARRAW .................................................................. 3-14
  CHAR ............................................................................. 3-14
    Input ............................................................................. 3-15
    Output .......................................................................... 3-15
  CHARZ ............................................................................ 3-15
  Named Data Types: Object, VARRAY, Nested Table .......... 3-16
  REF ................................................................................ 3-16
  ROWID Descriptor ............................................................. 3-16
  LOB Descriptor ................................................................. 3-17
    BFILE ........................................................................... 3-18
    BLOB ............................................................................ 3-18
    CLOB ............................................................................ 3-18
    NCLOB .......................................................................... 3-19
  Datetime and Interval Data Type Descriptors ...................... 3-19
### 4 Using SQL Statements in OCI

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of SQL Statement Processing</td>
<td>4-1</td>
</tr>
<tr>
<td>Preparing Statements</td>
<td>4-3</td>
</tr>
<tr>
<td>Using Prepared Statements on Multiple Servers</td>
<td>4-4</td>
</tr>
<tr>
<td>Binding Placeholders in OCI</td>
<td>4-4</td>
</tr>
<tr>
<td>Rules for Placeholders</td>
<td>4-5</td>
</tr>
<tr>
<td>Executing Statements</td>
<td>4-5</td>
</tr>
<tr>
<td>Execution Snapshots</td>
<td>4-6</td>
</tr>
<tr>
<td>Execution Modes of OCIStmtExecute()</td>
<td>4-6</td>
</tr>
<tr>
<td>Batch Error Mode</td>
<td>4-7</td>
</tr>
<tr>
<td>Example of Batch Error Mode</td>
<td>4-8</td>
</tr>
<tr>
<td>Describing Select-List Items</td>
<td>4-9</td>
</tr>
<tr>
<td>Implicit Describe</td>
<td>4-10</td>
</tr>
<tr>
<td>Explicit Describe of Queries</td>
<td>4-11</td>
</tr>
<tr>
<td>Defining Output Variables in OCI</td>
<td>4-12</td>
</tr>
<tr>
<td>Fetching Results</td>
<td>4-13</td>
</tr>
<tr>
<td>Fetching LOB Data</td>
<td>4-13</td>
</tr>
<tr>
<td>Setting Prefetch Count</td>
<td>4-13</td>
</tr>
<tr>
<td>Using Scorable Cursors in OCI</td>
<td>4-14</td>
</tr>
<tr>
<td>Increasing Scorable Cursor Performance</td>
<td>4-15</td>
</tr>
<tr>
<td>Example of Access on a Scorable Cursor</td>
<td>4-15</td>
</tr>
</tbody>
</table>

### 5 Binding and Defining in OCI

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of Binding in OCI</td>
<td>5-1</td>
</tr>
</tbody>
</table>
Character Conversion in OCI Binding and Defining DML with a RETURNING Clause in OCI

Advanced Bind Operations in OCI
    Binding LOBs
        Binding LOB Locators
        Binding LOB Data
    Binding in OCI_DATA_AT_EXEC Mode
    Binding REF CURSOR Variables

Overview of Defining in OCI
    Steps Used in OCI Defining

Advanced Define Operations in OCI
    Defining LOB Output Variables
    Defining LOB Locators
    Defining LOB Data
    Defining PL/SQL Output Variables
    Defining for a Piecewise Fetch

Binding and Defining Arrays of Structures in OCI
    Skip Parameters
        Skip Parameters for Standard Arrays
        OCI Calls Used with Arrays of Structures
        Arrays of Structures and Indicator Variables

Binding and Defining Multiple Buffers
    DML with a RETURNING Clause in OCI
        Using DML with a RETURNING Clause to Combine Two SQL Statements
        OCI Error Handling
        DML with RETURNING REF...INTO Clause in OCI
            Binding the Output Variable
        Additional Notes About OCI Callbacks
        Array Interface for DML RETURNING Statements in OCI

Character Conversion in OCI Binding and Defining
    Choosing a Character Set
        Character Set Form and ID
        Implicit Conversion Between CHAR and NCHAR
    Setting Client Character Sets in OCI
    Binding Variables in OCI
        Using the OCI_ATTR_MAXDATA_SIZE Attribute
        Using the OCI_ATTR_MAXCHAR_SIZE Attribute
        Buffer Expansion During OCI Binding
        Constraint Checking During Defining
        General Compatibility Issues for Character-Length Semantics in OCI

PL/SQL REF CURSORs and Nested Tables in OCI
    Valid Data Types for Piecewise Operations
6 Describing Schema Metadata

Using OCIDescribeAny() ........................................................................................................... 6-1
  Limitations on OCIDescribeAny() ......................................................................................... 6-2
  Notes on Types and Attributes ............................................................................................... 6-3
    Data Type Codes ..................................................................................................................... 6-3
    Describing Types .................................................................................................................. 6-3
    Implicit and Explicit Describe Operations ............................................................................. 6-3
    OCI_ATTR_LIST_ARGUMENTS Attribute .............................................................................. 6-4

Parameter Attributes ............................................................................................................... 6-4
  Table or View Parameters ....................................................................................................... 6-5
  Procedure, Function, and Subprogram Attributes .................................................................. 6-6
  Package Attributes .................................................................................................................. 6-6
  Type Attributes ...................................................................................................................... 6-7
  Type Attribute Attributes ....................................................................................................... 6-8
  Type Method Attributes ......................................................................................................... 6-9
  Collection Attributes ............................................................................................................ 6-10
  Synonym Attributes ............................................................................................................... 6-10
  Sequence Attributes .............................................................................................................. 6-11
  Column Attributes .................................................................................................................. 6-11
  Argument and Result Attributes ............................................................................................ 6-12
  List Attributes ....................................................................................................................... 6-14
  Schema Attributes ................................................................................................................. 6-14
  Database Attributes ............................................................................................................... 6-15
  Rule Attributes ...................................................................................................................... 6-15
  Rule Set Attributes ................................................................................................................. 6-16
  Evaluation Context Attributes ............................................................................................... 6-16
  Table Alias Attributes .......................................................................................................... 6-16
  Variable Type Attributes ....................................................................................................... 6-17
  Name Value Attributes .......................................................................................................... 6-17

Character-Length Semantics Support in Describe Operations .................................................. 6-17
  Implicit Describing ................................................................................................................. 6-18
  Explicit Describing ................................................................................................................. 6-18

Client and Server Compatibility Issues for Describing ............................................................. 6-18

Examples Using OCIDescribeAny() ......................................................................................... 6-18
  Retrieving Column Data Types for a Table ............................................................................ 6-19
  Describing the Stored Procedure ......................................................................................... 6-20
  Retrieving Attributes of an Object Type .............................................................................. 6-22
7 LOB and BFILE Operations

Using OCI Functions for LOBs .......................................................... 7-1
Creating and Modifying Persistent LOBs ....................................... 7-2
Associating a BFILE in a Table with an Operating System File .... 7-2
LOB Attributes of an Object ......................................................... 7-3
    Writing to a LOB Attribute of an Object ................................. 7-3
    Transient Objects with LOB Attributes ................................. 7-3
Array Interface for LOBs ......................................................... 7-3
Using LOBs of Size Greater than 4 GB ..................................... 7-4
    Functions to Use for the Increased LOB Sizes ...................... 7-5
    Compatibility and Migration ............................................. 7-6
LOB and BFILE Functions in OCI ............................................. 7-8
    Improving LOB Read/Write Performance ............................. 7-8
        Using Data Interface for LOBs ...................................... 7-9
        Using OCILobGetChunkSize() ...................................... 7-9
        Using OCILobWriteAppend2() ...................................... 7-9
        Using OCILobArrayRead() and OCILobArrayWrite() .......... 7-9
    LOB Buffering Functions ................................................. 7-9
    Functions for Opening and Closing LOBs ............................ 7-10
        Restrictions on Opening and Closing LOBs ................. 7-10
    LOB Read and Write Callbacks ....................................... 7-11
        Callback Interface for Streaming ................................. 7-11
        Reading LOBs by Using Callbacks ............................... 7-11
        Writing LOBs by Using Callbacks ............................... 7-13
Temporary LOB Support ...................................................... 7-14
    Creating and Freeing Temporary LOBs ............................... 7-15
    Temporary LOB Durations ............................................... 7-15
    Freeing Temporary LOBs ............................................... 7-16
    Take Care When Assigning Pointers .................................. 7-16
    Temporary LOB Example ............................................... 7-16
Prefetching of LOB Data, Length, and Chunk Size .................... 7-19
Options of SecureFiles LOBs .............................................. 7-22

8 Managing Scalable Platforms

OCI Support for Transactions ...................................................... 8-1
Levels of Transactional Complexity ............................................. 8-2
    Simple Local Transactions ............................................. 8-2
    Serializable or Read-Only Local Transactions ................. 8-2
    Global Transactions .................................................... 8-2
        Transaction Identifiers ............................................ 8-2
            Attribute OCI_ATTR_TRANS_NAME .......................... 8-3
        Transaction Branches ............................................. 8-3
        Branch States ...................................................... 8-4
    Detaching and Resuming Branches ..................................... 8-4
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixing 7.x and Later Release OCI Calls</td>
<td>8-26</td>
</tr>
<tr>
<td><strong>OCIThread Package</strong></td>
<td>8-26</td>
</tr>
<tr>
<td>Initialization and Termination</td>
<td>8-27</td>
</tr>
<tr>
<td>OCIThread Context</td>
<td>8-28</td>
</tr>
<tr>
<td>Passive Threading Primitives</td>
<td>8-28</td>
</tr>
<tr>
<td>OCIThreadMutex</td>
<td>8-28</td>
</tr>
<tr>
<td>OCIThreadKey</td>
<td>8-28</td>
</tr>
<tr>
<td>OCIThreadKeyDestFunc</td>
<td>8-28</td>
</tr>
<tr>
<td>OCIThreadId</td>
<td>8-29</td>
</tr>
<tr>
<td>Active Threading Primitives</td>
<td>8-29</td>
</tr>
<tr>
<td>OCIThreadHandle</td>
<td>8-30</td>
</tr>
<tr>
<td><strong>OCI Programming Advanced Topics</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Connection Pooling in OCI</strong></td>
<td></td>
</tr>
<tr>
<td>OCI Connection Pooling Concepts</td>
<td>9-1</td>
</tr>
<tr>
<td>Similarities and Differences from a Shared Server</td>
<td>9-2</td>
</tr>
<tr>
<td>Stateless Sessions Versus Stateful Sessions</td>
<td>9-2</td>
</tr>
<tr>
<td>Multiple Connection Pools</td>
<td>9-3</td>
</tr>
<tr>
<td>Transparent Application Failover</td>
<td>9-3</td>
</tr>
<tr>
<td>OCI Calls for Connection Pooling</td>
<td>9-3</td>
</tr>
<tr>
<td>Allocate the Pool Handle</td>
<td>9-4</td>
</tr>
<tr>
<td>Create the Connection Pool</td>
<td>9-4</td>
</tr>
<tr>
<td>Log On to the Database</td>
<td>9-5</td>
</tr>
<tr>
<td>Deal with SGA Limitations in Connection Pooling</td>
<td>9-6</td>
</tr>
<tr>
<td>Log Off from the Database</td>
<td>9-6</td>
</tr>
<tr>
<td>Destroy the Connection Pool</td>
<td>9-7</td>
</tr>
<tr>
<td>Free the Pool Handle</td>
<td>9-7</td>
</tr>
<tr>
<td>Examples of OCI Connection Pooling</td>
<td>9-7</td>
</tr>
<tr>
<td><strong>Session Pooling in OCI</strong></td>
<td></td>
</tr>
<tr>
<td>Functionality of OCI Session Pooling</td>
<td>9-8</td>
</tr>
<tr>
<td>Homogeneous and Heterogeneous Session Pool</td>
<td>9-8</td>
</tr>
<tr>
<td>Using Tags in Session Pools</td>
<td>9-9</td>
</tr>
<tr>
<td>OCI Handles for Session Pooling</td>
<td>9-9</td>
</tr>
<tr>
<td>OCISPool</td>
<td>9-9</td>
</tr>
<tr>
<td>OCIAuthInfo</td>
<td>9-9</td>
</tr>
<tr>
<td>Using OCI Session Pooling</td>
<td>9-10</td>
</tr>
<tr>
<td>OCI Calls for Session Pooling</td>
<td>9-10</td>
</tr>
<tr>
<td>Allocate the Pool Handle</td>
<td>9-11</td>
</tr>
<tr>
<td>Create the Pool Session</td>
<td>9-11</td>
</tr>
<tr>
<td>Log On to the Database</td>
<td>9-11</td>
</tr>
<tr>
<td>Log Off from the Database</td>
<td>9-12</td>
</tr>
<tr>
<td>Destroy the Session Pool</td>
<td>9-12</td>
</tr>
<tr>
<td>Free the Pool Handle</td>
<td>9-12</td>
</tr>
<tr>
<td>Example of OCI Session Pooling</td>
<td>9-12</td>
</tr>
<tr>
<td>Runtime Connection Load Balancing</td>
<td>9-12</td>
</tr>
<tr>
<td>Receiving Load Balancing Advisory FAN Events</td>
<td>9-13</td>
</tr>
<tr>
<td><strong>Database Resident Connection Pooling</strong></td>
<td>9-13</td>
</tr>
</tbody>
</table>
HA Event Notification ............................................................................................................................ 9-43
OCI Event Handle .................................................................................................................................... 9-44
OCI Failover for Connection and Session Pools .................................................................................... 9-44
OCI Failover for Independent Connections ......................................................................................... 9-45
Event Callback ....................................................................................................................................... 9-45
Custom Pooling: Tagged Server Handles .............................................................................................. 9-46
Determining Transparent Application Failover (TAF) Capabilities ....................................................... 9-47

OCI and Streams Advanced Queuing ..................................................................................................... 9-47
OCI Streams Advanced Queuing Functions ............................................................................................ 9-48
OCI Streams Advanced Queuing Descriptors ....................................................................................... 9-48
Streams Advanced Queuing in OCI Versus PL/SQL ................................................................................ 9-49
Buffered Messaging ............................................................................................................................... 9-52

Publish-Subscribe Notification in OCI ................................................................................................ 9-54
Publish-Subscribe Registration Functions in OCI .................................................................................. 9-55
Publish-Subscribe Register Directly to the Database ............................................................................ 9-55
Open Registration for Publish-Subscribe ............................................................................................... 9-58
Using OCI to Open Register with LDAP ............................................................................................... 9-59
Setting QOS, Timeout Interval, Namespace, Client Address, and Port Number ........................... 9-60
OCI Functions Used to Manage Publish-Subscribe Notification .......................................................... 9-61
Notification Callback in OCI .................................................................................................................. 9-61
Notification Procedure .......................................................................................................................... 9-64
Publish-Subscribe Direct Registration Example .................................................................................... 9-64
Publish-Subscribe LDAP Registration Example ..................................................................................... 9-69

10 More OCI Advanced Topics

Continuous Query Notification .............................................................................................................. 10-1
Using Query Result Set Notifications ................................................................................................... 10-2
Registering for Continuous Query Notification ..................................................................................... 10-2
Subscription Handle Attributes for Continuous Query Notification ................................................ 10-3
Using OCI_ATTR_CQ_QUERYID Attribute ........................................................................................... 10-5
Continuous Query Notification Descriptors .......................................................................................... 10-5
OCI_DTYPE_CHDES ............................................................................................................................ 10-6
Continuous Query Notification Example .............................................................................................. 10-7

Database Startup and Shutdown ........................................................................................................ 10-17
Examples of Startup and Shutdown in OCI ........................................................................................... 10-18

Implicit Fetching of ROWIDs .............................................................................................................. 10-20
Example of Implicit Fetching of ROWIDs .............................................................................................. 10-21

Client Result Cache ............................................................................................................................... 10-23
Benefits of Client Result Cache ............................................................................................................ 10-23
Guidelines for Using Client Result Cache ............................................................................................ 10-24
SQL Hints .............................................................................................................................................. 10-25
Table Annotation .................................................................................................................................. 10-25
Checking Table Annotation Mode ......................................................................................................... 10-26
Session Parameters ............................................................................................................................... 10-26
Effective Result Cache Table Mode ....................................................................................................... 10-26
Cache Example Use Cases ..................................................................................................................... 10-27
Queries That Are Not Cached ................................................................................................................. 10-27
Client Cache Consistency .................................................................................................................. 10-27
Deployment Time Settings for Client Result Cache .............................................................................. 10-28
Client Configuration File ........................................................................................................................ 10-29
Client Cache Statistics ............................................................................................................................ 10-29
Validation of the Client Result Cache ..................................................................................................... 10-30
Timing Measurement .............................................................................................................................. 10-30
Using v$mystat ..................................................................................................................................... 10-30
Using v$sqlarea ..................................................................................................................................... 10-30
OCI Client-Side Result Cache and Server Result Cache ...................................................................... 10-31
Client Result Cache Demo Files ............................................................................................................. 10-31
Compatibility with Previous Releases .................................................................................................. 10-32
Fault Diagnosability in OCI ................................................................................................................... 10-32
ADR Base Location ................................................................................................................................. 10-32
Using ADRCI ......................................................................................................................................... 10-33
Controlling ADR Creation and Disabling Fault Diagnosability Using sqlnet.ora ................................. 10-35
Client and Server Operating with Different Versions of Time Zone Files ......................................... 10-36

11 OCI Object-Relational Programming

OCI Object Overview ............................................................................................................................. 11-1
Working with Objects in OCI ................................................................................................................. 11-2
Basic Object Program Structure .......................................................................................................... 11-2
Persistent Objects, Transient Objects, and Values .............................................................................. 11-3
Persistent Objects ............................................................................................................................... 11-3
Transient Objects ............................................................................................................................... 11-4
Values .................................................................................................................................................. 11-4

Developing an OCI Object Application ............................................................................................ 11-5
Representing Objects in C Applications ............................................................................................... 11-5
Initializing the Environment and the Object Cache ............................................................................ 11-6
Making Database Connections .......................................................................................................... 11-7
Retrieving an Object Reference from the Server ................................................................................ 11-7
Pinning an Object ............................................................................................................................... 11-8
Array Pin ............................................................................................................................................. 11-9
Manipulating Object Attributes .......................................................................................................... 11-9
Marking Objects and Flushing Changes ............................................................................................... 11-10
Fetching Embedded Objects ................................................................................................................. 11-11
Object Meta-Attributes ........................................................................................................................ 11-12
Persistent Object Meta-Attributes ........................................................................................................ 11-12
Additional Attribute Functions ............................................................................................................ 11-14
Transient Object Meta-Attributes ........................................................................................................ 11-14
Complex Object Retrieval .................................................................................................................... 11-15
Prefetching Objects ............................................................................................................................. 11-16
Implementing Complex Object Retrieval in OCI ............................................................................. 11-17
COR Prefetching ................................................................................................................................. 11-17
COR Interface ....................................................................................................................................... 11-17
Example of COR ................................................................................................................................. 11-17
OCI Versus SQL Access to Objects ..................................................................................................... 11-18
Pin Count and Unpinning ..................................................................................................................... 11-18
Nested Table Element Ordering ................................................................. 12-17
Nested Table Locators .............................................................................. 12-17
Multilevel Collection Types ........................................................................ 12-17
Multilevel Collection Type Example ............................................................ 12-18
REF (OCIRef) ............................................................................................... 12-18
REF Manipulation Functions ........................................................................ 12-19
REF Example ............................................................................................... 12-19
Object Type Information Storage and Access ............................................... 12-20
Descriptor Objects ....................................................................................... 12-20
AnyType, AnyData, and AnyDataSet Interfaces ............................................ 12-20
Type Interfaces ............................................................................................ 12-21
Creating a Parameter Descriptor for OCIType Calls .................................... 12-21
Obtaining the OCIType for Persistent Types ................................................. 12-22
Type Access Calls ....................................................................................... 12-23
Extensions to OCIDescribeAny() .................................................................. 12-23
OCIAAnyData Interfaces ............................................................................. 12-23
NCHAR Typecodes for OCIAAnyData Functions ........................................... 12-24
OCIAAnyDataSet Interfaces ....................................................................... 12-24
Binding Named Data Types ....................................................................... 12-25
Named Data Type Binds ............................................................................... 12-25
Binding REFs ............................................................................................... 12-25
Information for Named Data Type and REF Binds ....................................... 12-26
Information Regarding Array Binds ............................................................. 12-26
Defining Named Data Types ..................................................................... 12-26
Defining Named Data Type Output Variables .............................................. 12-26
Defining REF Output Variables ................................................................... 12-27
Information for Named Data Type and REF Defines, and PL/SQL OUT Binds.................................................................................. 12-27
Information About Array Defines .............................................................. 12-28
Binding and Defining Oracle C Data Types .................................................. 12-28
Bind and Define Examples ........................................................................... 12-29
Salary Update Examples ............................................................................. 12-31
Method 1 - Fetch, Convert, Assign ............................................................... 12-32
Method 2 - Fetch and Assign ....................................................................... 12-32
Method 3 - Direct Fetch ............................................................................... 12-32
Summary and Notes .................................................................................... 12-33
SQLT_NTY Bind and Define Examples ......................................................... 12-33
SQLT_NTY Bind Example ............................................................................ 12-33
SQLT_NTY Define Example ........................................................................ 12-34

13 Direct Path Loading

Direct Path Loading Overview .................................................................... 13-1
Data Types Supported for Direct Path Loading ........................................... 13-3
Direct Path Handles ..................................................................................... 13-4
Direct Path Context ..................................................................................... 13-4
OCI Direct Path Function Context ............................................................... 13-4
Direct Path Column Array and Direct Path Function Column Array ............ 13-5
Direct Path Stream ..................................................................................... 13-6
Direct Path Interface Functions ................................................................. 13-6
Limitations and Restrictions of the Direct Path Load Interface .................... 13-7
Direct Path Load Examples for Scalar Columns ........................................... 13-7
Data Structures Used in Direct Path Loading Example ................................. 13-7
Outline of an Example of a Direct Path Load for Scalar Columns ................ 13-9
Using a Date Cache in Direct Path Loading of Dates in OCI .......................... 13-13
OCI_ATTR_DIRPATH_DCACHE_SIZE ......................................................... 13-13
OCI_ATTR_DIRPATH_DCACHE_NUM ......................................................... 13-13
OCI_ATTR_DIRPATH_DCACHE_MISSES .................................................. 13-13
OCI_ATTR_DIRPATH_DCACHE_HIT ................................................................ 13-14
OCI_ATTR_DIRPATH_DCACHE_DISABLE .................................................. 13-14
Direct Path Loading of Object Types .......................................................... 13-14
Direct Path Loading of Nested Tables .......................................................... 13-14
Describing a Nested Table Column and Its Nested Table ............................ 13-15
Direct Path Loading of Column Objects ....................................................... 13-15
Describing a Column Object ....................................................................... 13-15
Allocating the Array Column for the Column Object .................................... 13-17
Loading Column Object Data into the Column Array ................................... 13-17
OCI_DIRPATH_COL_ERROR ................................................................. 13-18
Direct Path Loading of SQL String Columns ................................................. 13-18
Describing a SQL String Column ............................................................... 13-18
Allocating the Column Array for SQL String Columns ................................. 13-20
Loading the SQL String Data into the Column Array .................................... 13-20
Direct Path Loading of REF Columns .......................................................... 13-21
Describing the REF Column ...................................................................... 13-21
Allocating the Column Array for a REF Column ......................................... 13-23
Loading the REF Data into the Column Array ............................................ 13-23
Direct Path Loading of NOT FINAL Object and REF Columns .................... 13-24
Inheritance Hierarchy .................................................................................. 13-24
Describing a Fixed, Derived Type to Be Loaded .......................................... 13-25
Allocating the Column Array ...................................................................... 13-25
Loading the Data into the Column Array .................................................... 13-25
Direct Path Loading of Object Tables .......................................................... 13-25
Direct Path Loading a NOT FINAL Object Table ......................................... 13-26
Direct Path Loading in Pieces .................................................................... 13-27
Loading Object Types in Pieces ............................................................... 13-27
Direct Path Context Handles and Attributes for Object Types .................... 13-28
Direct Path Context Attributes .................................................................. 13-28
OCI_ATTR_DIRPATH_OBJ_CONSTR ....................................................... 13-28
Direct Path Function Context and Attributes ............................................ 13-28
OCI_ATTR_DIRPATH_OBJ_CONSTR ....................................................... 13-28
OCI_ATTR_NAME ................................................................................... 13-29
OCI_ATTR_DIRPATH_EXPR_TYPE ............................................................ 13-30
OCI_ATTR_DIRPATH_NO_INDEX_ERRORS ................................................ 13-30
OCI_ATTR_NUM_COLS .......................................................................... 13-30
OCI_ATTR_NUM_ROWS .......................................................................... 13-31
Direct Path Column Parameter Attributes .................................................. 13-32
14 Object Advanced Topics in OCI

Object Cache and Memory Management ........................................ 14-1
  Cache Consistency and Coherency ........................................... 14-3
  Object Cache Parameters ....................................................... 14-4
  Object Cache Operations ...................................................... 14-4
    Pinning and Unpinning ....................................................... 14-4
    Freeing ................................................................................ 14-4
    Marking and Unmarking .................................................... 14-5
    Flushing ............................................................................... 14-5
    Refreshing ........................................................................... 14-5
  Loading and Removing Object Copies ................................... 14-5
    Pinning an Object Copy ...................................................... 14-5
    Unpinning an Object Copy ................................................ 14-7
    Freeing an Object Copy ..................................................... 14-7
  Making Changes to Object Copies ......................................... 14-7
    Marking an Object Copy .................................................... 14-7
    Unmarking an Object Copy ............................................... 14-8
  Synchronizing Object Copies with the Server .................... 14-8
    Flushing Changes to the Server ........................................ 14-8
    Refreshing an Object Copy ................................................. 14-9
  Object Locking ................................................................. 14-10
    Lock Options ....................................................................... 14-10
    Locking Objects for Update ............................................. 14-10
    Locking with the NOWAIT Option .................................... 14-10
    Implementing Optimistic Locking ...................................... 14-11
  Commit and Rollback in Object Applications ...................... 14-11
  Object Duration ............................................................... 14-11
    Durations Example .......................................................... 14-12
  Memory Layout of an Instance ............................................. 14-13

Object Navigation ................................................................. 14-14
  Simple Object Navigation .................................................. 14-14

OCI Navigational Functions .................................................... 14-15
  Pin/Unpin/Free Functions .................................................... 14-15
  Flush and Refresh Functions ............................................... 14-16
  Mark and Unmark Functions .............................................. 14-16
  Object Meta-Attribute Accessor Functions ......................... 14-16
  Other Functions .................................................................. 14-16

Type Evolution and the Object Cache ..................................... 14-17

OCI Support for XML ............................................................ 14-17
  XML Context ....................................................................... 14-18
  XML Data on the Server .................................................... 14-18
15 Using the Object Type Translator with OCI

OTT Overview .................................................................................................................. 15-1
What Is the Object Type Translator? .................................................................................. 15-2
Creating Types in the Database .......................................................................................... 15-3
Invoking OTT .................................................................................................................. 15-4
OTT Command Line ........................................................................................................... 15-4
OTT Command-Line Invocation Example ........................................................................... 15-5
OTT ....................................................................................................................................... 15-5
USERID ............................................................................................................................... 15-5
INTYPE ................................................................................................................................. 15-5
OUTTYPE ............................................................................................................................... 15-5
CODE .................................................................................................................................... 15-5
HFILE ..................................................................................................................................... 15-6
INITFILE ............................................................................................................................... 15-6
Intype File ........................................................................................................................... 15-6
OTT Data Type Mappings .................................................................................................... 15-8
Mapping Object Data Types to C ....................................................................................... 15-8
OTT Type Mapping Example ............................................................................................... 15-10
Null Indicator Structs ............................................................................................................. 15-12
OTT Support for Type Inheritance ....................................................................................... 15-13
Substitutable Object Attributes ............................................................................................ 15-15
Outtype File .......................................................................................................................... 15-15
Using OTT with OCI Applications ....................................................................................... 15-16
Accessing and Manipulating Objects with OCI ................................................................. 15-17
Calling the Initialization Function ....................................................................................... 15-18
Tasks of the Initialization Function ....................................................................................... 15-19
OTT Reference ...................................................................................................................... 15-19
OTT Command-Line Syntax ............................................................................................... 15-20
OTT Parameters .................................................................................................................... 15-20
USERID ................................................................................................................................. 15-21
INTYPE ................................................................................................................................. 15-22
OUTTYPE ............................................................................................................................... 15-22
CODE ..................................................................................................................................... 15-22
INITFILE ............................................................................................................................... 15-22
16 OCI Relational Functions

Introduction to the Relational Functions .......................................................... 16-1
Conventions for OCI Functions ........................................................................ 16-1
Purpose ............................................................................................................... 16-1
Syntax .............................................................................................................. 16-1
Parameters ....................................................................................................... 16-1
Comments ...................................................................................................... 16-2
Returns ........................................................................................................... 16-2
Example .......................................................................................................... 16-2
Related Functions ......................................................................................... 16-2
Calling OCI Functions ................................................................................... 16-2
Server Round-Trips for LOB Functions ......................................................... 16-2

Connect, Authorize, and Initialize Functions .................................................. 16-3
OCIAppCtxClearAll() .................................................................................. 16-4
OCIAppCtxSet() .......................................................................................... 16-5
OCIConnectionPoolCreate() ....................................................................... 16-7
OCIConnectionPoolDestroy() ...................................................................... 16-9
OCIDBShutdown() ....................................................................................... 16-10
OCIDBStartup() .......................................................................................... 16-12
OCIEnvCreate() .......................................................................................... 16-13
OCIEnvNlsCreate() .................................................................................... 16-17
OCILogoff() .................................................................................................. 16-21
OCILogon() ................................................................................................... 16-22
OCILogon2() ............................................................................................... 16-24
OCIServerAttach() ..................................................................................... 16-27
OCIServerDetach() ..................................................................................... 16-29
OCIComboBoxBegin() ................................................................................. 16-30
OCIComboBoxEnd() ..................................................................................... 16-33
OCISessionGet() .......................................................................................... 16-34
OCISessionPoolCreate() .......................................................................... 16-40

Where OTT Parameters Can Appear ............................................................... 15-25
Structure of the Intype File .......................................................................... 15-25
Intype File Type Specifications ................................................................. 15-26
Nested Included File Generation ............................................................... 15-27
SCHEMA_NAMES Usage ............................................................................. 15-29
Example: Schema_Names Usage .............................................................. 15-29
Default Name Mapping ............................................................................. 15-30
OTT Restriction on File Name Comparison .................................................. 15-31
17 More OCI Relational Functions

Introduction to the Relational Functions ................................................................. 17-1
Conventions for OCI Functions .............................................................................. 17-1

Statement Functions ............................................................................................... 17-2
OCIStmtExecute().................................................................................................. 17-3
OCIStmtFetch2().................................................................................................. 17-6
OCIStmtGetPieceInfo()......................................................................................... 17-8
OCIStmtPrepare()............................................................................................... 17-10
OCIStmtPrepare2()............................................................................................ 17-12
OCIStmtRelease()............................................................................................... 17-14
OCIStmtSetPieceInfo()....................................................................................... 17-15

LOB Functions ....................................................................................................... 17-17
OCI_DURATIONBEGIN()........................................................................................... 17-20
OCI_DURATION_END()............................................................................................. 17-21
OCILOB_APPEND().................................................................................................. 17-22
OCILOB_ARRAY_READ()......................................................................................... 17-24
OCILOB_ARRAY_WRITE()....................................................................................... 17-28
OCILOB_ASSIGN().................................................................................................. 17-32
OCILOB_CHARSET_FORM().................................................................................... 17-34
OCILOB_CHARSET_ID().......................................................................................... 17-35

Handle and Descriptor Functions .......................................................................... 16-47
OCIArrayDescriptorAlloc()................................................................................... 16-48
OCIArrayDescriptorFree()..................................................................................... 16-50
OCIAttrGet()....................................................................................................... 16-51
OCIAttrSet()....................................................................................................... 16-53
OCI_Descriptor_Alloc().......................................................................................... 16-54
OCI_Descriptor_Free().......................................................................................... 16-56
OCI_HANDLE_ALLOC()............................................................................................ 16-57
OCI_HANDLE_FREE()............................................................................................. 16-58
OCI_PARAM_GET()................................................................................................ 16-59
OCI_PARAM_SET()................................................................................................ 16-61

Bind, Define, and Describe Functions .................................................................. 16-62
OCI_BIND_ARRAY_OF_STRUCT()................................................................................ 16-63
OCI_BIND_BY_NAME()........................................................................................... 16-64
OCI_BIND_BY_POS()............................................................................................ 16-68
OCI_BIND_DYNAMIC()........................................................................................... 16-72
OCI_BIND_OBJECT()............................................................................................... 16-75
OCI_DEFINE_ARRAY_OF_STRUCT().......................................................................... 16-77
OCI_DEFINE_BY_POS()........................................................................................ 16-78
OCI_DEFINE_DYNAMIC()....................................................................................... 16-82
OCI_DEFINE_OBJECT().......................................................................................... 16-84
OCIgetDescription()............................................................................................ 16-86
OCI_STMT_GET_BIND_INFO()................................................................................ 16-89

OCISessionPoolDestroy()...................................................................................... 16-43
OCISessionRelease()........................................................................................... 16-44
OCI_Terminate()................................................................................................... 16-46
OCITerminate().................................................................................................... 16-46

OCISessionPoolDestroy()...................................................................................... 16-43
OCISessionRelease()........................................................................................... 16-44
OCITerminate().................................................................................................... 16-46
OCILobClose() .............................................................................................................................. 17-36
OCILobCopy2() ............................................................................................................................... 17-37
OCILobCreateTemporary() ............................................................................................................. 17-39
OCILobDisableBuffering() .............................................................................................................. 17-41
OCILobEnableBuffering() ................................................................................................................ 17-42
OCILobErase2() ............................................................................................................................... 17-43
OCILobFileClose() .......................................................................................................................... 17-45
OCILobFileCloseAll() ...................................................................................................................... 17-46
OCILobFileExists() ......................................................................................................................... 17-47
OCILobFileGetLength() .................................................................................................................. 17-48
OCILobFileIsOpen() ........................................................................................................................ 17-50
OCILobFileOpen() ............................................................................................................................ 17-51
OCILobFlushBuffer() ....................................................................................................................... 17-52
OCILobFreeTemporary() .................................................................................................................. 17-53
OCILobGetChunkSize() .................................................................................................................... 17-54
OCILobGetContentType() ................................................................................................................ 17-55
OCILobGetLength2() ....................................................................................................................... 17-56
OCILobGetOptions() ........................................................................................................................ 17-57
OCILobGetOptions() ....................................................................................................................... 17-58
OCILobGetStorageLimit() ................................................................................................................ 17-59
OCILobIsEqual() ................................................................................................................................ 17-60
OCILobIsTemporary() ....................................................................................................................... 17-61
OCILobLocateAssign() ..................................................................................................................... 17-62
OCILobLocateOpen() ......................................................................................................................... 17-63
OCILobLocateOpen() ......................................................................................................................... 17-64
OCILobLoadFromFile2() .................................................................................................................. 17-65
OCILobLocatorAssign() ................................................................................................................... 17-66
OCILobLocatorIsInit() ...................................................................................................................... 17-67
OCILobOpen() .................................................................................................................................. 17-68
OCILobRead2() .................................................................................................................................. 17-69
OCILobRead2() .................................................................................................................................. 17-70
OCILobSetContentType() .................................................................................................................. 17-71
OCILobSetOptions() ........................................................................................................................ 17-72
OCILobTrim2() .................................................................................................................................. 17-73
OCILobWrite2() .................................................................................................................................. 17-74
OCILobWriteAppend2() ..................................................................................................................... 17-75
 Streams Advanced Queuing and Publish-Subscribe Functions ................................................................ 17-76
                             OCIAQDeq() ................................................................................................................................ 17-77
                             OCIAQDeqArray() .................................................................................................................... 17-78
                             OCIAQEnq() ................................................................................................................................ 17-79
                             OCIAQEnqArray() ..................................................................................................................... 17-80
                             OCIAQListen2() ......................................................................................................................... 17-81
                             OCISubscriptionDisable() ....................................................................................................... 17-82
                             OCISubscriptionEnable() .......................................................................................................... 17-83
                             OCISubscriptionPost() .............................................................................................................. 17-84
                             OCISubscriptionRegister() .................................................................................................... 17-85
                             OCISubscriptionUnRegister() .................................................................................................. 17-86
 Direct Path Loading Functions ........................................................................................................ 17-87
                             OCIPathAbort() ................................................................................................................................ 17-88
                             OCIPathColArrayEntryGet() ..................................................................................................... 17-89
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIClientVersion()</td>
<td>17-156</td>
</tr>
<tr>
<td>OCITransStart()</td>
<td>17-154</td>
</tr>
<tr>
<td>OCITransRollback()</td>
<td>17-155</td>
</tr>
<tr>
<td>OCITransPrepare()</td>
<td>17-153</td>
</tr>
<tr>
<td>OCITransMultiPrepare()</td>
<td>17-152</td>
</tr>
<tr>
<td>OCITransForget()</td>
<td>17-151</td>
</tr>
<tr>
<td>OCITransCommit()</td>
<td>17-150</td>
</tr>
<tr>
<td>OCIThreadTerm()</td>
<td>17-149</td>
</tr>
<tr>
<td>OCIThreadProcessInit()</td>
<td>17-148</td>
</tr>
<tr>
<td>OCIThreadMutexRelease()</td>
<td>17-147</td>
</tr>
<tr>
<td>OCIThreadMutexInit()</td>
<td>17-146</td>
</tr>
<tr>
<td>OCIThreadKeySet()</td>
<td>17-145</td>
</tr>
<tr>
<td>OCIThreadKeyInit()</td>
<td>17-144</td>
</tr>
<tr>
<td>OCIThreadKeyGet()</td>
<td>17-143</td>
</tr>
<tr>
<td>OCIThreadJoin()</td>
<td>17-142</td>
</tr>
<tr>
<td>OCIThreadIsMulti()</td>
<td>17-141</td>
</tr>
<tr>
<td>OCIThreadDestroy()</td>
<td>17-140</td>
</tr>
<tr>
<td>OCIThreadDestroy()</td>
<td>17-139</td>
</tr>
<tr>
<td>OCIThreadInit()</td>
<td>17-138</td>
</tr>
<tr>
<td>OCIThreadSet()</td>
<td>17-137</td>
</tr>
<tr>
<td>OCIThreadIdNull()</td>
<td>17-136</td>
</tr>
<tr>
<td>OCIThreadIdSame()</td>
<td>17-135</td>
</tr>
<tr>
<td>OCIThreadIdSet()</td>
<td>17-134</td>
</tr>
<tr>
<td>OCIThreadHandleGet()</td>
<td>17-133</td>
</tr>
<tr>
<td>OCIThreadClose()</td>
<td>17-132</td>
</tr>
<tr>
<td>OCIDirPathColArraySet()</td>
<td>17-131</td>
</tr>
<tr>
<td>OCIDirPathColArrayReset()</td>
<td>17-130</td>
</tr>
<tr>
<td>OCIDirPathColArrayRowGet()</td>
<td>17-129</td>
</tr>
<tr>
<td>OCIDirPathColArrayRowSet()</td>
<td>17-128</td>
</tr>
<tr>
<td>OCIDirPathColArrayFlushRow()</td>
<td>17-127</td>
</tr>
<tr>
<td>OCIDirPathColArrayEntrySet()</td>
<td>17-126</td>
</tr>
<tr>
<td>OCIDirPathColArrayReset()</td>
<td>17-125</td>
</tr>
<tr>
<td>OCIDirPathColArrayRowGet()</td>
<td>17-124</td>
</tr>
<tr>
<td>OCIDirPathColArrayRowSet()</td>
<td>17-123</td>
</tr>
<tr>
<td>OCIDirPathColArrayFlushRow()</td>
<td>17-122</td>
</tr>
<tr>
<td>OCIDirPathColArrayEntrySet()</td>
<td>17-121</td>
</tr>
<tr>
<td>OCIDirPathColArrayReset()</td>
<td>17-120</td>
</tr>
<tr>
<td>OCIDirPathFinish()</td>
<td>17-119</td>
</tr>
<tr>
<td>OCIDirPathFlushRow()</td>
<td>17-118</td>
</tr>
<tr>
<td>OCIDirPathLoadStream()</td>
<td>17-117</td>
</tr>
<tr>
<td>OCIDirPathPrepare()</td>
<td>17-116</td>
</tr>
<tr>
<td>OCIDirPathStreamReset()</td>
<td>17-115</td>
</tr>
<tr>
<td>OCIDirPathColArrayEntrySet()</td>
<td>17-114</td>
</tr>
<tr>
<td>OCIDirPathColArrayReset()</td>
<td>17-113</td>
</tr>
<tr>
<td>OCIDirPathColArrayRowGet()</td>
<td>17-112</td>
</tr>
<tr>
<td>OCIDirPathColArrayRowSet()</td>
<td>17-111</td>
</tr>
<tr>
<td>OCIDirPathColArrayFlushRow()</td>
<td>17-110</td>
</tr>
<tr>
<td>OCIDirPathColArrayEntrySet()</td>
<td>17-109</td>
</tr>
</tbody>
</table>

**Thread Management Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIThreadClose()</td>
<td>17-133</td>
</tr>
<tr>
<td>OCIThreadCreate()</td>
<td>17-132</td>
</tr>
<tr>
<td>OCIThreadDestroy()</td>
<td>17-131</td>
</tr>
<tr>
<td>OCIThreadHandleGet()</td>
<td>17-130</td>
</tr>
<tr>
<td>OCIThreadHndDestroy()</td>
<td>17-129</td>
</tr>
<tr>
<td>OCIThreadIdGet()</td>
<td>17-128</td>
</tr>
<tr>
<td>OCIThreadIdDestroy()</td>
<td>17-127</td>
</tr>
<tr>
<td>OCIThreadIdSet()</td>
<td>17-126</td>
</tr>
<tr>
<td>OCIThreadIdSetNull()</td>
<td>17-125</td>
</tr>
<tr>
<td>OCIThreadInit()</td>
<td>17-124</td>
</tr>
<tr>
<td>OCIThreadIsMulti()</td>
<td>17-123</td>
</tr>
<tr>
<td>OCIThreadJoin()</td>
<td>17-122</td>
</tr>
<tr>
<td>OCIThreadKeyDestroy()</td>
<td>17-121</td>
</tr>
<tr>
<td>OCIThreadKeyGet()</td>
<td>17-120</td>
</tr>
<tr>
<td>OCIThreadKeyInit()</td>
<td>17-119</td>
</tr>
<tr>
<td>OCIThreadKeySet()</td>
<td>17-118</td>
</tr>
<tr>
<td>OCIThreadMutexAcquire()</td>
<td>17-117</td>
</tr>
<tr>
<td>OCIThreadMutexDestroy()</td>
<td>17-116</td>
</tr>
<tr>
<td>OCIThreadMutexInit()</td>
<td>17-115</td>
</tr>
<tr>
<td>OCIThreadMutexRelease()</td>
<td>17-114</td>
</tr>
<tr>
<td>OCIThreadProcessInit()</td>
<td>17-113</td>
</tr>
<tr>
<td>OCIThreadTerm()</td>
<td>17-112</td>
</tr>
</tbody>
</table>

**Transaction Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCITransCommit()</td>
<td>17-109</td>
</tr>
<tr>
<td>OCITransAttach()</td>
<td>17-111</td>
</tr>
<tr>
<td>OCITransForget()</td>
<td>17-112</td>
</tr>
<tr>
<td>OCITransMultiPrepare()</td>
<td>17-113</td>
</tr>
<tr>
<td>OCITransPrepare()</td>
<td>17-114</td>
</tr>
<tr>
<td>OCITransRollback()</td>
<td>17-115</td>
</tr>
<tr>
<td>OCITransStart()</td>
<td>17-116</td>
</tr>
</tbody>
</table>

**Miscellaneous Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIOpen()</td>
<td>17-158</td>
</tr>
<tr>
<td>OCIIdentify()</td>
<td>17-157</td>
</tr>
<tr>
<td>OCIServer()</td>
<td>17-156</td>
</tr>
<tr>
<td>OCIServerConnection()</td>
<td>17-155</td>
</tr>
<tr>
<td>OCIServerOpen()</td>
<td>17-154</td>
</tr>
<tr>
<td>OCIServerClose()</td>
<td>17-153</td>
</tr>
<tr>
<td>OCIServerAttach()</td>
<td>17-152</td>
</tr>
<tr>
<td>OCIServerInit()</td>
<td>17-151</td>
</tr>
<tr>
<td>OCIServerSet()</td>
<td>17-150</td>
</tr>
<tr>
<td>OCIServerCurVersion()</td>
<td>17-149</td>
</tr>
<tr>
<td>OCIServerGetCurrent()</td>
<td>17-148</td>
</tr>
<tr>
<td>OCIServerSetCurrent()</td>
<td>17-147</td>
</tr>
<tr>
<td>OCIServerSetVersion()</td>
<td>17-146</td>
</tr>
<tr>
<td>OCIServerGetVersion()</td>
<td>17-145</td>
</tr>
<tr>
<td>OCIServerStatus()</td>
<td>17-144</td>
</tr>
<tr>
<td>OCIServerError()</td>
<td>17-143</td>
</tr>
<tr>
<td>OCIServerStats()</td>
<td>17-142</td>
</tr>
<tr>
<td>OCIServerBusy()</td>
<td>17-141</td>
</tr>
<tr>
<td>OCIServerPing()</td>
<td>17-140</td>
</tr>
<tr>
<td>OCIServerPingCount()</td>
<td>17-139</td>
</tr>
<tr>
<td>OCIServerPingResult()</td>
<td>17-138</td>
</tr>
<tr>
<td>OCIServerPingError()</td>
<td>17-137</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-136</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-135</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-134</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-133</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-132</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-131</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-130</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-129</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-128</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-127</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-126</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-125</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-124</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-123</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-122</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-121</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-120</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-119</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-118</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-117</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-116</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-115</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-114</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-113</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-112</td>
</tr>
<tr>
<td>OCIServerPingBusy()</td>
<td>17-111</td>
</tr>
<tr>
<td>OCIServerPingStats()</td>
<td>17-110</td>
</tr>
</tbody>
</table>
### OCI Navigational and Type Functions

#### Introduction to the Navigational and Type Functions

Object Types and Lifetimes: 18-1
Terminology: 18-3

Conventions for OCI Functions: 18-3
Return Values: 18-3
Navigational Function Return Values: 18-3
Server Round-Trips for Cache and Object Functions: 18-3
Navigational Function Error Codes: 18-4

#### OCI Flush or Refresh Functions

OCICacheFlush(): 18-7
OCICacheRefresh(): 18-9
OCIObjectFlush(): 18-11
OCIObjectRefresh(): 18-12

#### OCI Mark or Unmark Object and Cache Functions

OCICacheUnmark(): 18-15
OCIObjectMarkDelete(): 18-16
OCIObjectMarkDeleteByRef(): 18-17
OCIObjectMarkUpdate(): 18-18
OCIObjectUnmark(): 18-19
OCIObjectUnmarkByRef(): 18-20

#### OCI Get Object Status Functions

OCIObjectExists(): 18-22
OCIObjectGetProperty(): 18-23
OCIObjectIsDirty(): 18-26
OCIObjectIsLocked(): 18-27

#### OCI Miscellaneous Object Functions

OCIObjectCopy(): 18-29
OCIObjectGetAttr(): 18-31
OCIObjectGetInd(): 18-33
OCIObjectGetObjectRef(): 18-34
OCIObjectGetTypeRef(): 18-35
OCIObjectLock(): 18-36
OCIObjectLockNoWait(): 18-37
OCIObjectNew(): 18-38
OCI Pin, Unpin, and Free Functions .......................................................... 18-43
OCICacheFree() .......................................................................................... 18-44
OCICacheUnpin() ......................................................................................... 18-45
OCIObjectArrayPin() .................................................................................. 18-46
OCIObjectFree() .......................................................................................... 18-48
OCIObjectPin() ............................................................................................ 18-50
OCIObjectPinCountReset() ....................................................................... 18-52
OCIObjectPinTable() .................................................................................. 18-53
OCIObjectUnpin() ......................................................................................... 18-55

OCI Type Information Accessor Functions ................................................. 18-57
OCITypeArrayByName() ............................................................................. 18-58
OCITypeArrayByRef() ................................................................................ 18-60
OCITypeByName() ....................................................................................... 18-62
OCITypeByRef() .......................................................................................... 18-64

19 OCI Data Type Mapping and Manipulation Functions

Introduction to Data Type Mapping and Manipulation Functions .................. 19-1
Conventions for OCI Functions ................................................................. 19-1
Returns .......................................................................................................... 19-2
Data Type Mapping and Manipulation Function Return Values .................. 19-2
Functions Returning Other Values ............................................................ 19-2
Server Round-Trips for Data Type Mapping and Manipulation Functions ...... 19-2
Examples ...................................................................................................... 19-2

OCI Collection and Iterator Functions ......................................................... 19-3
OCI CollAppend() ....................................................................................... 19-4
OCI CollAssign() ......................................................................................... 19-5
OCI CollAssignElem() ................................................................................. 19-6
OCI CollGetElem() ...................................................................................... 19-7
OCI CollGetElemArray() ............................................................................. 19-10
OCI CollIsLocator() .................................................................................. 19-12
OCI CollMax() ........................................................................................... 19-13
OCI CollSize() ............................................................................................ 19-14
OCI CollTrim() ............................................................................................ 19-16
OCIIterCreate() ........................................................................................ 19-17
OCIIterDelete() .......................................................................................... 19-18
OCIIterGetCurrent() ................................................................................... 19-19
OCIIterInit() ............................................................................................... 19-20
OCIIterNext() ............................................................................................. 19-21
OCIIterPrev() ............................................................................................. 19-23

OCI Date, Datetime, and Interval Functions ................................................. 19-25
OCIDateAddDays() ..................................................................................... 19-27
OCIDateAddMonths() ................................................................................. 19-28
OCIDateAssign() ....................................................................................... 19-29
OCIDateCheck() ........................................................................................ 19-30
OCIDateCompare() ..................................................................................... 19-32
OCIDateDaysBetween() ............................................................................. 19-33
OCI NUMBER Functions

- OCIIntervalToText()
- OCIIntervalToNumber()
- OCIIntervalSetYearMonth()
- OCIIntervalSetDaySecond()
- OCIIntervalGetYearMonth()
- OCIIntervalGetDaySecond()
- OCINumberArcCos()
- OCINumberAbs()
- OCIIntervalFromTZ()
- OCIIntervalFromText()
- OCIIntervalFromNumber()
- OCIIntervalCompare()
- OCIIntervalAssign()
- OCIIntervalAdd()
- OCIDateZoneToZone()
- OCIDateToText()
- OCIDateTimeToText()
- OCIDateTimeToArray()
- OCIDateTimeSysTimeStamp()
- OCIDateTimeSubtract()
- OCIDateTimeIntervalSub()
- OCIDateTimeGetTimeZoneOffset()
- OCIDateTimeGetTimeZoneName()
- OCIDateTimeGetDate()
- OCIDateTimeFromText()
- OCIDateTimeConvert()
- OCIDateTimeCompare()
- OCIDateTimeAssign()
- OCIDateSysDate()
- OCIDateSetTime()
- OCIDateSetDate()
- OCIDateNextDay()
- OCIDateGetTime()
- OCIDateGetDate()
- OCIDateFromText()
- OCIIntervalFromTZ()
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIAssignRaw()</td>
<td>19-94</td>
</tr>
<tr>
<td>OCIRawAssignBytes()</td>
<td>19-95</td>
</tr>
<tr>
<td>OCIRawAllocSize()</td>
<td>19-96</td>
</tr>
<tr>
<td>OCINumberTrunc()</td>
<td>19-97</td>
</tr>
<tr>
<td>OCINumberToText()</td>
<td>19-98</td>
</tr>
<tr>
<td>OCINumberToReal()</td>
<td>19-99</td>
</tr>
<tr>
<td>OCINumberToInt()</td>
<td>19-100</td>
</tr>
<tr>
<td>OCINumberTan()</td>
<td>19-101</td>
</tr>
<tr>
<td>OCINumberSqrt()</td>
<td>19-102</td>
</tr>
<tr>
<td>OCINumberSign()</td>
<td>19-103</td>
</tr>
<tr>
<td>OCINumberShift()</td>
<td>19-104</td>
</tr>
<tr>
<td>OCINumberSetPi()</td>
<td>19-105</td>
</tr>
<tr>
<td>OCINumberPrec()</td>
<td>19-106</td>
</tr>
<tr>
<td>OCINumberPower()</td>
<td>19-107</td>
</tr>
<tr>
<td>OCINumberNeg()</td>
<td>19-108</td>
</tr>
<tr>
<td>OCINumberMul()</td>
<td>19-109</td>
</tr>
<tr>
<td>OCINumberMod()</td>
<td>19-110</td>
</tr>
<tr>
<td>OCINumberLog()</td>
<td>19-111</td>
</tr>
<tr>
<td>OCINumberIsZero()</td>
<td>19-112</td>
</tr>
<tr>
<td>OCINumberLn()</td>
<td>19-113</td>
</tr>
<tr>
<td>OCINumberLog()</td>
<td>19-114</td>
</tr>
<tr>
<td>OCINumberMod()</td>
<td>19-115</td>
</tr>
<tr>
<td>OCINumberMul()</td>
<td>19-116</td>
</tr>
<tr>
<td>OCINumberNeg()</td>
<td>19-117</td>
</tr>
<tr>
<td>OCINumberPower()</td>
<td>19-118</td>
</tr>
<tr>
<td>OCINumberPrec()</td>
<td>19-119</td>
</tr>
<tr>
<td>OCINumberRound()</td>
<td>19-120</td>
</tr>
<tr>
<td>OCINumberSetPi()</td>
<td>19-121</td>
</tr>
<tr>
<td>OCINumberSetZero()</td>
<td>19-122</td>
</tr>
<tr>
<td>OCINumberShift()</td>
<td>19-123</td>
</tr>
<tr>
<td>OCINumberSign()</td>
<td>19-124</td>
</tr>
<tr>
<td>OCINumberSin()</td>
<td>19-125</td>
</tr>
<tr>
<td>OCINumberSqrt()</td>
<td>19-126</td>
</tr>
<tr>
<td>OCINumberSub()</td>
<td>19-127</td>
</tr>
<tr>
<td>OCINumberTan()</td>
<td>19-128</td>
</tr>
<tr>
<td>OCINumberToInt()</td>
<td>19-129</td>
</tr>
<tr>
<td>OCINumberToReal()</td>
<td>19-130</td>
</tr>
<tr>
<td>OCINumberToRealArray()</td>
<td>19-131</td>
</tr>
<tr>
<td>OCINumberToText()</td>
<td>19-132</td>
</tr>
<tr>
<td>OCINumberTrunc()</td>
<td>19-133</td>
</tr>
</tbody>
</table>

**OCI Raw Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIAllocSize()</td>
<td>19-95</td>
</tr>
<tr>
<td>OCIAssignBytes()</td>
<td>19-96</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-97</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-98</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-99</td>
</tr>
<tr>
<td>OCIAllocSize()</td>
<td>19-100</td>
</tr>
<tr>
<td>OCIAssignBytes()</td>
<td>19-101</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-102</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-103</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-104</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-105</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-106</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-107</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-108</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-109</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-110</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-111</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-112</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-113</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-114</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-115</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-116</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-117</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-118</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-119</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-120</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-121</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-122</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-123</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-124</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-125</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-126</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-127</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-128</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-129</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-130</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-131</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-132</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-133</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-134</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-135</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-136</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-137</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-138</td>
</tr>
<tr>
<td>OCIAssignRaw()</td>
<td>19-139</td>
</tr>
</tbody>
</table>
20 OCI Cartridge Functions

Introduction to External Procedure and Cartridge Services Functions ........................................... 20-1
Conventions for OCI Functions ........................................................................................................... 20-1
Return Codes ...................................................................................................................................... 20-1
With_Context Type ............................................................................................................................ 20-2

Cartridge Services — OCI External Procedures .............................................................................. 20-3
OCIExtProcAllocCallMemory() .......................................................................................................... 20-4
OCIExtProcGetEnv() .......................................................................................................................... 20-5
OCIExtProcRaiseExcp() ................................................................................................................... 20-6
OCIExtProcRaiseExcpWithMsg() ....................................................................................................... 20-7

Cartridge Services — Memory Services .......................................................................................... 20-8
OCIDurationBegin() .......................................................................................................................... 20-9
OCIDurationEnd() .............................................................................................................................. 20-10
OCIMemoryAlloc() ............................................................................................................................ 20-11
OCIMemoryFree() ............................................................................................................................. 20-12
OCIMemoryResize() .......................................................................................................................... 20-13

Cartridge Services — Maintaining Context ...................................................................................... 20-14
OCIContextClearValue() .................................................................................................................... 20-15
OCIContextGenerateKey() .................................................................................................................. 20-16
OCIContextGetValue() ....................................................................................................................... 20-17
OCIContextSetValue() ....................................................................................................................... 20-18
21 OCI Any Type and Data Functions

Introduction to Any Type and Data Interfaces ............................................................... 21-1

Conventions for OCI Functions ....................................................................................... 21-1

Function Return Values .................................................................................................... 21-1

OCI Type Interface Functions .......................................................................................... 21-3

OCITypeAddAttr() ............................................................................................................. 21-4

OCITypeBeginCreate() ...................................................................................................... 21-5

OCITypeEndCreate() ......................................................................................................... 21-6

OCITypeSetBuiltin() .......................................................................................................... 21-7

OCITypeSetCollection() .................................................................................................... 21-8

OCI Any Data Interface Functions .................................................................................... 21-9

OCIAnyDataAccess() ......................................................................................................... 21-10

OCIAnyDataAttrGet() ....................................................................................................... 21-12

OCIAnyDataAttrSet() ........................................................................................................ 21-14

OCIAnyDataBeginCreate() ............................................................................................... 21-16

OCIAnyDataCollAddElem() .............................................................................................. 21-18

OCIAnyDataCollGetElem() ............................................................................................... 21-20
22 OCI Globalization Support Functions

Introduction to Globalization Support in OCI ......................................................... 22-1
Conventions for OCI Functions .............................................................................. 22-1
Returns...................................................................................................................... 22-1

OCI Locale Functions .............................................................................................. 22-3
OCINlsCharSetIdToName()..................................................................................... 22-4
OCINlsCharSetNameToId()..................................................................................... 22-5
OCINlsEnvironmentVariableGet()........................................................................ 22-6
OCINlsGetInfo()..................................................................................................... 22-8
OCINlsNumericInfoGet()....................................................................................... 22-11

OCI Locale-Mapping Function ............................................................................... 22-12
OCINlsNameMap().................................................................................................. 22-13

OCI String Manipulation Functions ...................................................................... 22-14
OCIMultiByteInSizeToWideChar()....................................................................... 22-16
OCIMultiByteStrCaseConversion()....................................................................... 22-17
OCIMultiByteStrcat()............................................................................................ 22-18
OCIMultiByteStrcmp().......................................................................................... 22-19
OCIMultiByteStrcpy().......................................................................................... 22-20
OCIMultiByteStrlen()............................................................................................ 22-21
OCIMultiByteStrncat().......................................................................................... 22-22
OCIMultiByteStrncmp()......................................................................................... 22-23
OCIMultiByteStrncpy()......................................................................................... 22-25
OCIMultiByteStrnDisplayLength()....................................................................... 22-26
OCIMultiByteToWideChar()................................................................................ 22-27
OCIWideCharInSizeToMultiByte()....................................................................... 22-28
OCIWideCharMultiByteLength().......................................................................... 22-29
OCIWideCharStrCaseConversion()...................................................................... 22-30
OCIWideCharStrcat()............................................................................................ 22-31
OCIWideCharStrchr()............................................................................................ 22-32
OCIWideCharStrcmp()........................................................................................... 22-33
OCIWideCharStrcpy()........................................................................................... 22-34
OCIWideCharStrlen() ................................................................. 22-35
OCIWideCharStrncmp() .......................................................... 22-37
OCIWideCharStrncpy() ......................................................... 22-39
OCIWideCharStrchr() ............................................................ 22-40
OCIWideCharToLower() .......................................................... 22-41
OCIWideCharToMultiByte() ................................................ 22-42
OCIWideCharToUpper() .......................................................... 22-43

OCI Character Classification Functions ........................................ 22-44
OCIWideCharIsAlnum() .......................................................... 22-45
OCIWideCharIsAlpha() ......................................................... 22-46
OCIWideCharIsCtrl() ............................................................. 22-47
OCIWideCharIsDigit() ........................................................... 22-48
OCIWideCharIsGraph() ......................................................... 22-49
OCIWideCharIsLower() .......................................................... 22-50
OCIWideCharIsPrint() ........................................................... 22-51
OCIWideCharIsPunct() ............................................................ 22-52
OCIWideCharIsSingleByte() .................................................. 22-53
OCIWideCharIsSpace() .......................................................... 22-54
OCIWideCharIsUpper() ........................................................... 22-55
OCIWideCharIsXDigit() .......................................................... 22-56

OCI Character Set Conversion Functions ...................................... 22-57
OCICltrSetConvert() ............................................................... 22-58
OCICltrSetToUnicode() ........................................................... 22-59
OCINlsCharSetConvert() ........................................................ 22-60
OCINlsCharSetToCharSet() ..................................................... 22-62

OCI Messaging Functions .......................................................... 22-63
OCIMessageClose() ............................................................... 22-64
OCIMessageGet() ................................................................. 22-65
OCIMessageOpen() ............................................................... 22-66

23 OCI XML DB Functions

Introduction to XML DB Support in OCI ........................................ 23-1
Conventions for OCI Functions ............................................... 23-1
Returns ................................................. 23-1

OCI XML DB Functions .......................................................... 23-2
OCI BinXmlCreateReposCtxFromConn() .................................. 23-3
OCI BinXmlCreateReposCtxFromCPool() .................................. 23-4
OCI BinXmlSetFormatPref() .................................................... 23-5
OCI BinXmlSetReposCtxForConn() ........................................... 23-6
OCI XmlDbFreeXmlCtx() ........................................................ 23-7
OCI XmlDbInitXmlCtx() ........................................................ 23-8

A Handle and Descriptor Attributes

Conventions ................................................................. A-2
Environment Handle Attributes ................................................. A-2
Error Handle Attributes ........................................................ A-8
<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Context Handle Attributes</td>
<td>A-8</td>
</tr>
<tr>
<td>Server Handle Attributes</td>
<td>A-11</td>
</tr>
<tr>
<td>Authentication Information Handle</td>
<td>A-14</td>
</tr>
<tr>
<td>User Session Handle Attributes</td>
<td>A-14</td>
</tr>
<tr>
<td>Administration Handle Attributes</td>
<td>A-22</td>
</tr>
<tr>
<td>Connection Pool Handle Attributes</td>
<td>A-22</td>
</tr>
<tr>
<td>Session Pool Handle Attributes</td>
<td>A-24</td>
</tr>
<tr>
<td>Transaction Handle Attributes</td>
<td>A-26</td>
</tr>
<tr>
<td>Statement Handle Attributes</td>
<td>A-27</td>
</tr>
<tr>
<td>Bind Handle Attributes</td>
<td>A-34</td>
</tr>
<tr>
<td>Define Handle Attributes</td>
<td>A-37</td>
</tr>
<tr>
<td>Describe Handle Attributes</td>
<td>A-39</td>
</tr>
<tr>
<td>Parameter Descriptor Attributes</td>
<td>A-39</td>
</tr>
<tr>
<td>LOB Locator Attributes</td>
<td>A-39</td>
</tr>
<tr>
<td>Complex Object Attributes</td>
<td>A-40</td>
</tr>
<tr>
<td>Complex Object Retrieval Handle Attributes</td>
<td>A-40</td>
</tr>
<tr>
<td>Complex Object Retrieval Descriptor Attributes</td>
<td>A-40</td>
</tr>
<tr>
<td>Streams Advanced Queuing Descriptor Attributes</td>
<td>A-41</td>
</tr>
<tr>
<td>OCIAQEnqOptions Descriptor Attributes</td>
<td>A-41</td>
</tr>
<tr>
<td>OCIAQReqOptions Descriptor Attributes</td>
<td>A-43</td>
</tr>
<tr>
<td>OCIAQMsgProperties Descriptor Attributes</td>
<td>A-46</td>
</tr>
<tr>
<td>OCIAQAgent Descriptor Attributes</td>
<td>A-50</td>
</tr>
<tr>
<td>OCIServerDNs Descriptor Attributes</td>
<td>A-51</td>
</tr>
<tr>
<td>Subscription Handle Attributes</td>
<td>A-51</td>
</tr>
<tr>
<td>Continuous Query Notification Attributes</td>
<td>A-57</td>
</tr>
<tr>
<td>Continuous Query Notification Descriptor Attributes</td>
<td>A-58</td>
</tr>
<tr>
<td>Notification Descriptor Attributes</td>
<td>A-60</td>
</tr>
<tr>
<td>Invalidated Query Attributes</td>
<td>A-62</td>
</tr>
<tr>
<td>Direct Path Loading Handle Attributes</td>
<td>A-62</td>
</tr>
<tr>
<td>Direct Path Context Handle (OCIDirPathCtx) Attributes</td>
<td>A-63</td>
</tr>
<tr>
<td>Direct Path Function Context Handle (OCIDirPathFuncCtx) Attributes</td>
<td>A-68</td>
</tr>
<tr>
<td>Direct Path Function Column Array Handle (OCIDirPathColArray) Attributes</td>
<td>A-69</td>
</tr>
<tr>
<td>Direct Path Stream Handle (OCIDirPathStream) Attributes</td>
<td>A-70</td>
</tr>
<tr>
<td>Direct Path Column Parameter Attributes</td>
<td>A-71</td>
</tr>
<tr>
<td>Accessing Column Parameter Attributes</td>
<td>A-71</td>
</tr>
<tr>
<td>Process Handle Attributes</td>
<td>A-75</td>
</tr>
<tr>
<td>Event Handle Attributes</td>
<td>A-77</td>
</tr>
</tbody>
</table>

**B** OCI Demonstration Programs

**C** OCI Function Server Round-Trips

<table>
<thead>
<tr>
<th>Round-Trips Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of Server Round-Trips</td>
<td>C-1</td>
</tr>
<tr>
<td>Relational Function Round-Trips</td>
<td>C-1</td>
</tr>
<tr>
<td>LOB Function Round-Trips</td>
<td>C-3</td>
</tr>
<tr>
<td>Object and Cache Function Round-Trips</td>
<td>C-4</td>
</tr>
<tr>
<td>Describe Operation Round-Trips</td>
<td>C-5</td>
</tr>
</tbody>
</table>
D Getting Started with OCI for Windows

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

E Deprecated OCI Functions

Deprecated Initialize Functions ................................................................................ E-2
    OCIEnvInit() ....................................................................................................... E-3
    OCIInitialize() .................................................................................................. E-5
Deprecated Statement Functions .............................................................................. E-7
    OCIStmtFetch() ............................................................................................... E-8
Deprecated Lob Functions ....................................................................................... E-9
    OCILobCopy() ................................................................................................. E-10
    OCILobErase() ............................................................................................... E-11
    OCILobGetLength() ....................................................................................... E-12
    OCILobLoadFromFile() .................................................................................. E-13
    OCILobRead() ............................................................................................... E-14
    OCILobTrim() ............................................................................................... E-18
    OCILobWrite() ............................................................................................... E-19
    OCILobWriteAppend() .................................................................................. E-23
Deprecated Streams Advanced Queuing Functions ................................................ E-26
    OCIAQListen() ............................................................................................... E-27

Index
List of Examples

2–1 Using the OCI_ATTR_USERNAME Attribute to Set the User Name in the Session Handle
2–8
2–2 Returning Describe Information in the Statement Handle Relating to Select-List Items
2–8
2–3 Using the OCIlogon2 Call for a Single User Session
2–15
2–4 Enabling a Local User to Serve as a Proxy for Another User
2–16
2–5 Connection String to Use for the Proxy User
2–16
2–6 Preserving Case Sensitivity When Enabling a Local User to Serve as a Proxy for Another User
2–16
2–7 Preserving Case Sensitivity in the Connection String
2–16
2–8 Using "dilbert[mybert]" in the Connection String
2–16
2–9 Using "dilbert[mybert]"["joe[myjoe"]" in the Connection String
2–17
2–10 Setting the Target User Name
2–17
2–11 Using OCI to Set the OCI_ATTR_PROXY_CLIENT Attribute and the Proxy dilbert...
2–17
2–12 Creating and Initializing an OCI Environment
2–18
2–13 Getting Locale Information in OCI
2–32
2–14 Basic String Manipulation in OCI
2–33
2–15 Classifying Characters in OCI
2–34
2–16 Converting Character Sets in OCI
2–35
2–17 Retrieving a Message from a Text Message File
2–36
3–1 OCI Bind and Define Support for 64-Bit Integers
3–10
3–2 Binding 8-Byte Integer Data Types for OUT Binds of a DML Returning Statement 3–10
4–1 Binding Both Input and Output Variables in Nonquery Operations
4–5
4–2 Calling OCIAttrGet() to Retrieve the Number of Errors Encountered During an Array DML Operation
4–7
4–3 Retrieving Information About Each Error Following an Array DML Operation
4–7
4–4 Using Batch Error Execution Mode
4–8
4–5 Implicit Describe - Select List Is Available as an Attribute of the Statement Handle
4–10
4–6 Explicit Describe - Returning the Select-List Description for Each Column
4–12
4–7 Access on a Scrollable Cursor
4–16
5–1 Handle Allocation and Binding for Each Placeholder in a SQL Statement
5–5
5–2 Defining a PL/SQL Statement to Be Used in OCI
5–6
5–3 Binding the Placeholder and Executing the Statement to Insert a Single Locator
5–8
5–4 Binding the Placeholder and Executing the Statement to Insert an Array of Locators
5–9
5–5 Demonstrating Some Implicit Conversions That Cannot Be Done
5–10
5–6 Allowed: Inserting into C1, C2, and L Columns Up to 8000, 8000, and 2000 Byte-Sized Bind Variable Data Values, Respectively
5–11
5–7 Allowed: Inserting into C1 and L Columns up to 2000 and 8000 Byte-Sized Bind Variable Data Values, Respectively
5–11
5–8 Allowed: Updating C1, C2, and L Columns up to 8000, 8000, and 2000 Byte-Sized Bind Variable Data Values, Respectively
5–11
5–9 Allowed: Updating C1, C2, and L Columns up to 2000, 2000, and 8000 Byte-Sized Bind Variable Data Values, Respectively
5–12
5–10 Allowed: Piecewise, Callback, and Array Insert or Update Operations
5–12
5–11 Not Allowed: Inserting More Than 4000 Bytes into Both LOB and LONG Columns Using the Same INSERT Statement
5–12
5–12 Allowed: Inserting into the CT3 LOB Column up to 2000 Byte-Sized Bind Variable Data Values
5–12
5–13 Not Allowed: Binding Any Length Data to a LOB Column in an Insert As Select Operation
5–13
5–14 Defining a Scalar Output Variable Following an Execute and Describe Operation
5–14
5–15 Defining LOBs Before Execution
5–17
5–16 Defining LOBs After Execution
5–17
5–17 Using Multiple Bind and Define Buffers
5–20
5–18 Binding the REF Output Variable in an OCI Application
5–25
5-19 Setting the Client Character Set to OCI_UTF16ID in OCI .................................................. 5-27
5-20 Insert and Select Operations Using the OCI_ATTR_MAXCHAR_SIZE Attribute .......... 5-31
5-21 Binding and Defining UTF-16 Data.................................................................................... 5-32
5-22 Binding the :cursor1 Placeholder to the Statement Handle stm2p as a REF CURSOR . . 5-33
5-23 Defining a Nested Table (Second Position) as a Statement Handle ......................... 5-33
6-1 Initializing the OCI Process in Object Mode................................................................. 6-3
6-2 Using an Explicit Describe to Retrieve Column Data Types for a Table ...................... 6-19
6-3 Describing the Stored Procedure.................................................................................. 6-20
6-4 Using an Explicit Describe on a Named Object Type................................................... 6-22
6-5 Using an Explicit Describe on a Named Collection Type.......................................... 6-23
6-6 Using a Parameter Descriptor to Retrieve the Data Types, Column Names, and Character-Length Semantics 6-25
7-1 Using the LOB Locator and Allocating the Descriptors................................................... 7-4
7-2 Implementing Read Callback Functions Using OCIlobRead2()..................................... 7-12
7-3 Implementing Write Callback Functions Using OCIlobWrite2().................................... 7-13
7-4 Using Temporary LOBs ............................................................................................... 7-16
7-5 Prefetching of LOB Data, Length, and Chunk Size....................................................... 7-20
8-1 Defining the OCI_ATTR_SERVER_GROUP Attribute to Pass the Server Group Name . 8-9
8-2 Defining the OCI_ATTR_PROXY_CREDENTIALS Attribute to Specify the Credentials of the Application Server for Client Authentication 8-11
8-3 Defining the OCI_ATTR_DISTINGUISHED_NAME Attribute to Pass the Distinguished Name of the Client 8-11
8-4 Defining the OCI_ATTR_CERTIFICATE Attribute to Pass the Entire X.509 Certificate .... 8-12
8-5 Defining the OCI_ATTR_INITIALIZIAL_CLIENT_ROLES Attribute to Pass the Client Roles .... 8-12
8-6 Defining the OCI_ATTR_CLIENT_IDENTIFIER Attribute to Pass the End-User Identity..... 8-12
8-7 Defining the OCI_ATTR_PASSWORD Attribute to Pass the Password for Validation ...... 8-13
8-8 OCI Attributes That Let You Specify the External Name and Initial Privileges of a Client ... 8-13
8-9 Defining the OCI_ATTR_APPCTX_SIZE Attribute to Initialize the Context Array Size with the Desired Number of Context Attributes 8-16
8-10 Using the OCI_ATTR_APPCTX_LIST Attribute to Get a Handle on the Application Context List Descriptor for the Session 8-16
8-11 Calling OCIParamGet() to Obtain an Individual Descriptor for the i-th Application Context Using the Application Context List Descriptor 8-16
8-12 Defining Session Handle Attributes to Set Externally Initialized Context............... 8-17
8-13 Using the OCI_ATTR_CALL_TIME Attribute to Get the Elapsed Time of the Last Server Call 8-17
8-14 Using OCISESSIONBEGIN() with an Externally Initialized Context.......................... 8-18
8-15 Changing the "responsibility" Attribute Value in the CLIENTCONTEXT Namespace 8-20
8-16 Two Ways to Clear Specific Attribute Information in a Client Namespace............... 8-21
8-17 Clearing All the Context Information in a Specific Client Namespace...................... 8-21
8-18 Calling OCIAttrSet() to Set the OCI_ATTR_EDITION Attribute ................................. 8-22
9-1 Setting Session Purity ................................................................................................. 9-17
9-2 Setting the Connection Class as HRMS..................................................................... 9-17
9-3 Setting the Connection Class as RECMS .................................................................... 9-17
9-4 Specifying :POOLED in the Easy Connect String for Enabling DRCP ....................... 9-18
9-5 Specifying SERVER=POOLED in a TNS Connect String for Enabling DRCP .............. 9-18
9-6 Database Resident Connection Pooling Application............................................... 9-19
9-7 Connect String to Use for a Deployment in Dedicated Server Mode with DRCP Not Enabled 9-20
9-8 Connect String to Use for a Deployment with DRCP Enabled ................................... 9-20
9-9 Optimizing Bind and Define Operations on Statements in the Cache....................... 9-29
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2–1</td>
<td>Basic OCI Program Flow</td>
<td>2-2</td>
</tr>
<tr>
<td>2–2</td>
<td>Components of a Service Context</td>
<td>2-5</td>
</tr>
<tr>
<td>2–3</td>
<td>Statement Handles</td>
<td>2-6</td>
</tr>
<tr>
<td>2–4</td>
<td>Direct Path Handles</td>
<td>2-8</td>
</tr>
<tr>
<td>4–1</td>
<td>Steps in Processing SQL Statements</td>
<td>4-2</td>
</tr>
<tr>
<td>5–1</td>
<td>Using OCIBindByName() to Associate Placeholders with Program Variables</td>
<td>5-2</td>
</tr>
<tr>
<td>5–2</td>
<td>Determining Skip Parameters</td>
<td>5-18</td>
</tr>
<tr>
<td>5–3</td>
<td>Performing Piecewise Insert</td>
<td>5-36</td>
</tr>
<tr>
<td>5–4</td>
<td>Performing Piecewise Fetch</td>
<td>5-39</td>
</tr>
<tr>
<td>6–1</td>
<td>OCIDescribeAny() Table Description</td>
<td>6-2</td>
</tr>
<tr>
<td>8–1</td>
<td>Multiple Tightly Coupled Branches</td>
<td>8-4</td>
</tr>
<tr>
<td>8–2</td>
<td>Session Operating on Multiple Branches</td>
<td>8-4</td>
</tr>
<tr>
<td>9–1</td>
<td>OCI Connection Pooling</td>
<td>9-3</td>
</tr>
<tr>
<td>9–2</td>
<td>Publish-Subscribe Model</td>
<td>9-54</td>
</tr>
<tr>
<td>11–1</td>
<td>Basic Object Operational Flow</td>
<td>11-5</td>
</tr>
<tr>
<td>13–1</td>
<td>Direct Path Loading</td>
<td>13-2</td>
</tr>
<tr>
<td>13–2</td>
<td>Inheritance Hierarchy for a Column of Type Person</td>
<td>13-24</td>
</tr>
<tr>
<td>14–1</td>
<td>Object Cache</td>
<td>14-3</td>
</tr>
<tr>
<td>14–2</td>
<td>Object Graph of person_t Instances</td>
<td>14-14</td>
</tr>
<tr>
<td>15–1</td>
<td>Using OTT with OCI</td>
<td>15-17</td>
</tr>
<tr>
<td>18–1</td>
<td>Classification of Instances by Type and Lifetime</td>
<td>18-2</td>
</tr>
</tbody>
</table>
# List of Tables

1-1 Obsolescent OCI Functions ................................................................. 1-13
1-2 OCI Functions Not Supported ............................................................ 1-15
1-3 OCI Instant Client Shared Libraries .................................................... 1-16
1-4 OCI Instant Client Light Shared Libraries ......................................... 1-25
2-1 OCI Handle Types ............................................................................. 2-3
2-2 Descriptor Types ............................................................................. 2-9
2-3 OCI Return Codes ........................................................................... 2-21
2-4 Return and Error Codes ................................................................. 2-21
2-5 Oracle Reserved Namespaces ........................................................... 2-26
3-1 Internal Oracle Database Data Types .............................................. 3-3
3-2 External Data Types and Codes ....................................................... 3-6
3-3 VARNUM Examples ......................................................................... 3-12
3-4 Format of the DATE Data Type ......................................................... 3-13
3-5 Data Conversions ............................................................................ 3-21
3-6 Data Conversions for LOBs .............................................................. 3-22
3-7 Data Conversion for Datetime and Interval Types ......................... 3-23
3-8 Data Conversion for External Data Types to Internal Numeric Data Types ......................................................... 3-24
3-9 Data Conversions for Internal to External Numeric Data Types ........ 3-25
3-10 OCITypeCode Values and Data Types ............................................ 3-26
3-11 OCI_TYPECODE to SQLT Mappings ............................................. 3-27
4-1 OCI_ATTR_STMT_TYPE Values and Statement Types ..................... 4-3
5-1 Information Summary for Bind Types ............................................ 5-7
6-1 Attributes of All Parameters ............................................................ 6-4
6-2 Attributes of Tables or Views ........................................................... 6-5
6-3 Attributes Specific to Tables ............................................................ 6-6
6-4 Attributes of Procedures or Functions ............................................ 6-6
6-5 Attributes Specific to Package Subprograms .................................... 6-6
6-6 Attributes of Packages ................................................................... 6-7
6-7 Attributes of Types .......................................................................... 6-7
6-8 Attributes of Type Attributes .......................................................... 6-8
6-9 Attributes of Type Methods ............................................................. 6-9
6-10 Attributes of Collection Types ...................................................... 6-10
6-11 Attributes of Synonyms .................................................................. 6-11
6-12 Attributes of Sequences ................................................................. 6-11
6-13 Attributes of Columns of Tables or Views ...................................... 6-12
6-14 Attributes of Arguments and Results ............................................. 6-13
6-15 List Attributes ................................................................................ 6-14
6-16 Attributes Specific to Schemas ....................................................... 6-15
6-17 Attributes Specific to Databases ..................................................... 6-15
6-18 Attributes Specific to Rules ............................................................ 6-16
6-19 Attributes Specific to Rule Sets ....................................................... 6-16
6-20 Attributes Specific to Evaluation Contexts ...................................... 6-16
6-21 Attributes Specific to Table Aliases .............................................. 6-17
6-22 Attributes Specific to Variable Types ............................................. 6-17
6-23 Attributes Specific to Name-Value Pair ......................................... 6-17
7-1 LOB Functions Compatibility and Migration ...................................... 7-6
8-1 Global Transaction Identifier .......................................................... 8-3
8-2 One-Phase Commit ......................................................................... 8-6
8-3 Two-Phase Commit .......................................................................... 8-7
8-4 Read-Only Update Fails ................................................................. 8-7
8-5 Read-Only Transaction ................................................................. 8-7
8-6 Initialization and Termination Multithreading Functions ................. 8-27
8-7 Passive Threading Primitives ......................................................... 8-28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8–8</td>
<td>Active Threading Primitives</td>
<td>8-30</td>
</tr>
<tr>
<td>9–1</td>
<td>Defaults Used in Various Client Scenarios</td>
<td>9-18</td>
</tr>
<tr>
<td>9–2</td>
<td>Time and Event</td>
<td>9-42</td>
</tr>
<tr>
<td>9–3</td>
<td>AQ Functions</td>
<td>9-49</td>
</tr>
<tr>
<td>9–4</td>
<td>Enqueue Parameters</td>
<td>9-49</td>
</tr>
<tr>
<td>9–5</td>
<td>Dequeue Parameters</td>
<td>9-49</td>
</tr>
<tr>
<td>9–6</td>
<td>Listen Parameters</td>
<td>9-50</td>
</tr>
<tr>
<td>9–7</td>
<td>Array Enqueue Parameters</td>
<td>9-50</td>
</tr>
<tr>
<td>9–8</td>
<td>Array Dequeue Parameters</td>
<td>9-50</td>
</tr>
<tr>
<td>9–9</td>
<td>Agent Parameters</td>
<td>9-50</td>
</tr>
<tr>
<td>9–10</td>
<td>Message Properties</td>
<td>9-51</td>
</tr>
<tr>
<td>9–11</td>
<td>Enqueue Option Attributes</td>
<td>9-51</td>
</tr>
<tr>
<td>9–12</td>
<td>Dequeue Option Attributes</td>
<td>9-51</td>
</tr>
<tr>
<td>9–13</td>
<td>Publish-Subscribe Functions</td>
<td>9-61</td>
</tr>
<tr>
<td>10–1</td>
<td>DDL Table Result Cache Annotation Modes</td>
<td>10-25</td>
</tr>
<tr>
<td>10–2</td>
<td>Effective Result Cache Table Mode</td>
<td>10-26</td>
</tr>
<tr>
<td>10–3</td>
<td>Setting Client-Side Result Cache and Server Result Cache</td>
<td>10-31</td>
</tr>
<tr>
<td>11–1</td>
<td>Meta-Attributes of Persistent Objects</td>
<td>11-12</td>
</tr>
<tr>
<td>11–2</td>
<td>Set and Check Functions</td>
<td>11-14</td>
</tr>
<tr>
<td>11–3</td>
<td>Transient Meta-Attributes</td>
<td>11-14</td>
</tr>
<tr>
<td>11–4</td>
<td>Attribute Values for New Objects</td>
<td>11-24</td>
</tr>
<tr>
<td>12–1</td>
<td>Function Prefix Examples</td>
<td>12-4</td>
</tr>
<tr>
<td>12–2</td>
<td>Binding and Defining Datetime and Interval Data Types</td>
<td>12-6</td>
</tr>
<tr>
<td>12–3</td>
<td>Datetime Functions</td>
<td>12-7</td>
</tr>
<tr>
<td>12–4</td>
<td>Interval Functions</td>
<td>12-9</td>
</tr>
<tr>
<td>12–5</td>
<td>String Functions</td>
<td>12-12</td>
</tr>
<tr>
<td>12–6</td>
<td>Raw Functions</td>
<td>12-13</td>
</tr>
<tr>
<td>12–7</td>
<td>Collection Functions</td>
<td>12-14</td>
</tr>
<tr>
<td>12–8</td>
<td>Collection Scanning Functions</td>
<td>12-15</td>
</tr>
<tr>
<td>12–9</td>
<td>Nested Table Functions</td>
<td>12-16</td>
</tr>
<tr>
<td>12–10</td>
<td>REF Manipulation Functions</td>
<td>12-19</td>
</tr>
<tr>
<td>12–11</td>
<td>Descriptor Objects</td>
<td>12-20</td>
</tr>
<tr>
<td>12–12</td>
<td>Data Type Mappings for Binds and Defines</td>
<td>12-29</td>
</tr>
<tr>
<td>13–1</td>
<td>Direct Path Context Functions</td>
<td>13-6</td>
</tr>
<tr>
<td>13–2</td>
<td>Direct Path Column Array Functions</td>
<td>13-7</td>
</tr>
<tr>
<td>14–1</td>
<td>Object Attributes After a Refresh Operation</td>
<td>14-9</td>
</tr>
<tr>
<td>14–2</td>
<td>Example of Allocation and Pin Durations</td>
<td>14-12</td>
</tr>
<tr>
<td>14–3</td>
<td>Pin, Free, and Unpin Functions</td>
<td>14-15</td>
</tr>
<tr>
<td>14–4</td>
<td>Flush and Refresh Functions</td>
<td>14-16</td>
</tr>
<tr>
<td>14–5</td>
<td>Mark and Unmark Functions</td>
<td>14-16</td>
</tr>
<tr>
<td>14–6</td>
<td>Object Meta-Attributes Functions</td>
<td>14-16</td>
</tr>
<tr>
<td>14–7</td>
<td>Other Object Functions</td>
<td>14-17</td>
</tr>
<tr>
<td>15–1</td>
<td>Object Data Type Mappings for Object Type Attributes</td>
<td>15-9</td>
</tr>
<tr>
<td>16–1</td>
<td>Mode of a Parameter</td>
<td>16-2</td>
</tr>
<tr>
<td>16–2</td>
<td>Connect, Authorize, and Initialize Functions</td>
<td>16-3</td>
</tr>
<tr>
<td>16–3</td>
<td>Handle and Descriptor Functions</td>
<td>16-47</td>
</tr>
<tr>
<td>16–4</td>
<td>Bind, Define, and Describe Functions</td>
<td>16-62</td>
</tr>
<tr>
<td>17–1</td>
<td>Statement Functions</td>
<td>17-2</td>
</tr>
<tr>
<td>17–2</td>
<td>LOB Functions</td>
<td>17-17</td>
</tr>
<tr>
<td>17–3</td>
<td>Advanced Queuing and Publish-Subscribe Functions</td>
<td>17-88</td>
</tr>
<tr>
<td>17–4</td>
<td>Direct Path Loading Functions</td>
<td>17-106</td>
</tr>
<tr>
<td>17–5</td>
<td>Thread Management Functions</td>
<td>17-121</td>
</tr>
<tr>
<td>17–6</td>
<td>Transaction Functions</td>
<td>17-147</td>
</tr>
<tr>
<td>17–7</td>
<td>Miscellaneous Functions</td>
<td>17-162</td>
</tr>
<tr>
<td>17–8</td>
<td>OCI Function Codes</td>
<td>17-182</td>
</tr>
<tr>
<td>Page</td>
<td>Section</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>17–9</td>
<td>Continuation of OCI Function Codes from 97 and Higher</td>
<td></td>
</tr>
<tr>
<td>18–1</td>
<td>Type and Lifetime of Instances</td>
<td></td>
</tr>
<tr>
<td>18–2</td>
<td>Return Values of Navigational Functions</td>
<td></td>
</tr>
<tr>
<td>18–3</td>
<td>OCI Navigational Functions Error Codes</td>
<td></td>
</tr>
<tr>
<td>18–4</td>
<td>Flush or Refresh Functions</td>
<td></td>
</tr>
<tr>
<td>18–5</td>
<td>Object Status After Refresh</td>
<td></td>
</tr>
<tr>
<td>18–6</td>
<td>Mark or Unmark Object and Cache Functions</td>
<td></td>
</tr>
<tr>
<td>18–7</td>
<td>Get Object Status Functions</td>
<td></td>
</tr>
<tr>
<td>18–8</td>
<td>Miscellaneous Object Functions</td>
<td></td>
</tr>
<tr>
<td>18–9</td>
<td>Instances Created</td>
<td></td>
</tr>
<tr>
<td>18–10</td>
<td>Pin, Unpin, and Free Functions</td>
<td></td>
</tr>
<tr>
<td>18–11</td>
<td>Type Information Accessor Functions</td>
<td></td>
</tr>
<tr>
<td>19–1</td>
<td>Function Return Values</td>
<td></td>
</tr>
<tr>
<td>19–2</td>
<td>Collection and Iterator Functions</td>
<td></td>
</tr>
<tr>
<td>19–3</td>
<td>Element Pointers</td>
<td></td>
</tr>
<tr>
<td>19–4</td>
<td>Date Functions</td>
<td></td>
</tr>
<tr>
<td>19–5</td>
<td>Error Bits Returned by the valid Parameter for OCIDateCheck()</td>
<td></td>
</tr>
<tr>
<td>19–6</td>
<td>Comparison Results</td>
<td></td>
</tr>
<tr>
<td>19–7</td>
<td>Error Bits Returned by the valid Parameter for OCIDateTimeCheck()</td>
<td></td>
</tr>
<tr>
<td>19–8</td>
<td>Comparison Results Returned by the result Parameter for OCIDateTimeCompare()</td>
<td></td>
</tr>
<tr>
<td>19–9</td>
<td>Error Bits Returned by the valid Parameter for OCIIntervalCheck()</td>
<td></td>
</tr>
<tr>
<td>19–10</td>
<td>Comparison Results Returned by the result Parameter for OCIIntervalCompare()</td>
<td></td>
</tr>
<tr>
<td>19–11</td>
<td>NUMBER Functions</td>
<td></td>
</tr>
<tr>
<td>19–12</td>
<td>Comparison Results Returned by the result Parameter for OCINumberCmp()</td>
<td></td>
</tr>
<tr>
<td>19–13</td>
<td>Values of result</td>
<td></td>
</tr>
<tr>
<td>19–14</td>
<td>Raw Functions</td>
<td></td>
</tr>
<tr>
<td>19–15</td>
<td>Ref Functions</td>
<td></td>
</tr>
<tr>
<td>19–16</td>
<td>String Functions</td>
<td></td>
</tr>
<tr>
<td>19–17</td>
<td>Table Functions</td>
<td></td>
</tr>
<tr>
<td>20–1</td>
<td>External Procedures Functions</td>
<td></td>
</tr>
<tr>
<td>20–2</td>
<td>Memory Services Functions</td>
<td></td>
</tr>
<tr>
<td>20–3</td>
<td>Maintaining Context Functions</td>
<td></td>
</tr>
<tr>
<td>20–4</td>
<td>Parameter Manager Interface Functions</td>
<td></td>
</tr>
<tr>
<td>20–5</td>
<td>File I/O Interface Functions</td>
<td></td>
</tr>
<tr>
<td>20–6</td>
<td>String Formatting Functions</td>
<td></td>
</tr>
<tr>
<td>20–7</td>
<td>Format Modifier Flags</td>
<td></td>
</tr>
<tr>
<td>20–8</td>
<td>Format Codes to Specify How to Format an Argument Written to a String</td>
<td></td>
</tr>
<tr>
<td>21–1</td>
<td>Function Return Values</td>
<td></td>
</tr>
<tr>
<td>21–2</td>
<td>Type Interface Functions</td>
<td></td>
</tr>
<tr>
<td>21–3</td>
<td>Any Data Functions</td>
<td></td>
</tr>
<tr>
<td>21–4</td>
<td>Data Types and Attribute Values</td>
<td></td>
</tr>
<tr>
<td>21–5</td>
<td>Data Types and Attribute Values</td>
<td></td>
</tr>
<tr>
<td>21–6</td>
<td>Any Data Set Functions</td>
<td></td>
</tr>
<tr>
<td>22–1</td>
<td>Function Return Values</td>
<td></td>
</tr>
<tr>
<td>22–2</td>
<td>OCI Locale Functions</td>
<td></td>
</tr>
<tr>
<td>22–3</td>
<td>OCI Locale-Mapping Function</td>
<td></td>
</tr>
<tr>
<td>22–4</td>
<td>OCI String Manipulation Functions</td>
<td></td>
</tr>
<tr>
<td>22–5</td>
<td>OCI Character Classification Functions</td>
<td></td>
</tr>
<tr>
<td>22–6</td>
<td>OCI Character Set Conversion Functions</td>
<td></td>
</tr>
<tr>
<td>22–7</td>
<td>OCI Messaging Functions</td>
<td></td>
</tr>
<tr>
<td>23–1</td>
<td>Function Return Values</td>
<td></td>
</tr>
<tr>
<td>23–2</td>
<td>OCI XML DB Functions</td>
<td></td>
</tr>
<tr>
<td>A–1</td>
<td>SQL Command Codes</td>
<td></td>
</tr>
<tr>
<td>B–1</td>
<td>OCI Demonstration Programs</td>
<td></td>
</tr>
<tr>
<td>C–1</td>
<td>Server Round-Trips for Relational Operations</td>
<td></td>
</tr>
</tbody>
</table>
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

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 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 Streams Advanced Queuing User’s Guide supplies information about OCI calls pertaining to Advanced Queuing.
- Oracle Database Advanced Application Developer’s 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 Installation Guide for Microsoft Windows
- Oracle Database Release Notes for Microsoft Windows
- Oracle Database Net Services Administrator’s Guide
- Oracle Database New Features Guide
- Oracle Database Concepts
- Oracle Database Reference
- Oracle Database Error Messages

The Oracle C++ Call Interface provides OCI functionality for C++ programs and enables programmers to manipulate database objects of user-defined types as C++ objects. For more information about OCI functionality for C++, see the Oracle C++ Call Interface Programmer’s Guide

Conventions

The following text conventions are used in this document:
<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><strong>monospace</strong></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
What's New in Oracle Call Interface?

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

- New Features in Oracle Call Interface Release 11.2
- New Features in Oracle Call Interface Release 11.1

New Features in Oracle Call Interface Release 11.2

This release includes the following new or enhanced features:

- The client and server can use different time zone files.
  See "Client and Server Operating with Different Versions of Time Zone Files" on page 10-36.
- Application patching is improved with edition-based redefinition.
  See "Edition-Based Redefinition" on page 8-22.
- OCI supports 64-bit host data types.
  See "64-Bit Integer Host Data Type" on page 3-9.
- New attributes can specify IP address format as either IPv4 or IPv6.
- Content types for SecureFiles are used with hierarchical storage.
  See "OCILobGetContentType()" on page 17-56 and "OCILobSetContentType()" on page 17-77.
- There is a new direct path loading attribute.
  See "OCI_ATTR_DIRPATH_NO_INDEX_ERRORS" on page 13-30.
- The client result cache feature now has table annotations, and it supports caching on a query with views. Also, the restriction causing the client cache to be disabled with database resident pooling has been removed.
  See "Client Result Cache" on page 10-23.
- The process to regenerate the data shared library and the zip and RPM files has changed for Instant Client.
  See "Regeneration of Data Shared Library and Zip and RPM Files" on page 1-19.
- You can use OCI to access Oracle TimesTen In-Memory Database and Oracle In-Memory Database Cache.
New Features in Oracle Call Interface Release 11.1

The following new or enhanced features were introduced in release 11.1 of Oracle Call Interface:

- Run-time load balancing is available automatically and improves performance.
  See "Runtime Connection Load Balancing" on page 9-12.

- To improve performance, binds and defines now allow noncontiguous buffers for reading and writing (scatter and gather for binds and defines).
  See "Binding and Defining Multiple Buffers" on page 5-20.

- Use the `SELECT ... FOR UPDATE` statement to implicitly fetch ROWIDs.
  See "Implicit Fetching of ROWIDs" on page 10-20.

- Performance is improved by the use of arrays of descriptors.

- The client result cache feature can improve performance.
  See "Client Result Cache" on page 10-23.

- New settings provided for database administrators make OCI more secure.
  See "OCI Security Enhancements" on page 8-23.

- Support for XML DB and XDK documents has been enhanced to include binary XML.
  See "OCI Support for XML" on page 14-17.

- OCI applications can optimize fetching of length, chunk size, and data of small LOBs.
  See "Prefetching of LOB Data, Length, and Chunk Size" on page 7-19.

- SecureFiles LOBs. New capabilities of LOBs include encryption, deduplication, and compression. New OCI functions are included in this release. These new functions and features are only usable on the new SecureFiles LOBs.
  See "Options of SecureFiles LOBs" on page 7-22.

- Database resident connection pooling. The pool spans processes instead of just spanning different threads. More efficient applications can result.

- Database change notification enhancements for query result set changes are added to the preexisting capabilities.
  See "Continuous Query Notification" on page 10-1.

- Oracle Streams Advanced Queuing event notification now supports notifications spaced in time.
  See "Publish-Subscribe Notification in OCI" on page 9-54.

- Fault diagnostic subdirectory structures help users manage OCI-generated incidents.
  See "Fault Diagnosability in OCI" on page 10-32.
To read or write the client driver layer name to improve OCI fault diagnosability, use the following attribute on the session handle.


New version compatibility of OCI applications is described.

See “Version Compatibility of Statically Linked and Dynamically Linked Applications” on page 1-12.
This chapter contains these topics:

- Overview of OCI
- Compatibility and Upgrading
- OCI Instant Client

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

See Also:

- Oracle C++ Call Interface Programmer's Guide
- "Related Documents" on page 2-xlvi

OCI provides:

- Improved performance and scalability through the efficient use of system memory and network connectivity
- Consistent interfaces for dynamic session and transaction management in a two-tier client/server or multitier environment
- N-tier authentication
- Comprehensive support for application development using Oracle Database objects
- Access to external databases
- Applications that support an increasing number of users and requests without additional hardware investments

OCI enables you to manipulate data and schemas in an Oracle Database using the C programming language. It provides a library of standard database access and retrieval functions in the form of a dynamic runtime library (OCI library) that can be linked in an application at run time.

You can use OCI to access Oracle TimesTen In-Memory Database and Oracle In-Memory Database Cache. See Oracle TimesTen In-Memory Database C Developer’s Guide.
OCI has many new features that can be categorized into several primary areas:

- Encapsulated or opaque interfaces, whose implementation details are unknown
- Simplified user authentication and password management
- Extensions to improve application performance and scalability
- Consistent interface for transaction management
- OCI extensions to support client-side access to Oracle objects

**Advantages of OCI**

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

- More fine-grained control over all aspects of application design
- High degree of control over program execution
- Use of familiar third-generation language programming techniques and application development tools, such as browsers and debuggers
- Connection pooling, session pooling, and statement caching that enable building of scalable applications
- Support of dynamic SQL
- Availability on the broadest range of operating systems of all the Oracle programmatic interfaces
- Dynamic binding and defining using callbacks
- Description functionality to expose layers of server metadata
- Asynchronous event notification for registered client applications
- Enhanced array data manipulation language (DML) capability for array inserts, updates, and deletes
- Ability to associate commit requests with executes to reduce round-trips
- Optimization of queries using transparent prefetch buffers to reduce round-trips
- Thread safety, which eliminates the need for mutual exclusive locks (mutexes) on OCI handles

**Building an OCI Application**

You compile and link an OCI program in the same way that you compile and link a non-database application. There is no need for a separate preprocessing or precompilation step.

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

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

OCI has the following functionality:

- APIs to design a scalable, multithreaded application that can support large numbers of users securely
- SQL access functions, for managing database access, processing SQL statements, and manipulating objects retrieved from an Oracle database
- 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

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.

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.

See Also:

- Oracle Database Concepts
- Oracle Database Object-Relational Developer's Guide.

OCI includes functions that extend the capabilities of OCI to handle objects in an Oracle Database. 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

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

SQL Statements

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

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

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

■ PL/SQL
■ Embedded SQL

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

Data Definition Language

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

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

```
CREATE TABLE employees
  (name       VARCHAR2(20),
   ssn        VARCHAR2(12),
   empno      NUMBER(6),
   mgr        NUMBER(6),
   salary     NUMBER(6));
GRANT UPDATE, INSERT, DELETE ON employees TO donna;
REVOKE UPDATE ON employees FROM jamie;
```

DDL statements also allow you to work with objects in the Oracle database, 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;
```

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

Data Manipulation Language

Data manipulation language (DML) statements can change data in the database tables. For example, DML statements are used to:

■ Insert new rows into a table
■ Update column values in existing rows
■ Delete rows from a table
■ Lock a table in the database
■ Explain the execution plan for a SQL statement
■ Require an application to supply data to the database using input (bind) variables

See Also: "Binding Placeholders in OCI" on page 4-4 for more information about input bind variables
DML statements also allow you to work with objects in the Oracle database, as in the following example, which inserts an instance of type `person_t` into the object table `person_tab`:

```sql
INSERT INTO person_tab
    VALUES (person_t('Steve May', '987-65-4320', '146 Winfield Street'));
```

Queries

Queries are statements that retrieve data from a database. A query can return zero, one, or many rows of data. All queries begin with the SQL keyword `SELECT`, as in the following example:

```sql
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:

```sql
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" on page 5-1 for more information about input bind variables
- "Overview of Defining in OCI" on page 5-13 for information about defining output variables
- Chapter 4, for detailed information about how SQL statements are processed in an OCI program

PL/SQL

PL/SQL is Oracle's procedural extension to the SQL language. PL/SQL processes tasks that are more complicated than simple queries and SQL data manipulation language statements. PL/SQL allows 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
- 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.

```sql
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
- "Binding Placeholders in PL/SQL" on page 5-4 for information about working with placeholders in PL/SQL

**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*

**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:

```sql
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. These terms and operations are described in Chapter 4.

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

Simplified User Authentication and Password Management

OCI provides application developers with simplified user authentication and password management 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 simplified login function for sessions by which a single user connects to the database using a login name and password
- A mechanism by which a single OCI application authenticates and maintains multiple sessions by separating the login session (the session created when a user logs in to an Oracle database) from the user sessions (all other sessions created by a user)

Extensions 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:

- Increased client-side processing, and reduced server-side requirements on queries
- Implicit prefetching of \texttt{SELECT} statement result sets to eliminate the describe round-trip, reduce round-trips, and reduce memory usage
- Elimination of open and closed cursor round-trips
- Improved support for multithreaded environments
- Session multiplexing over connections
- Consistent support for a variety of configurations, including standard two-tier client/server configurations, server-to-server transaction coordination, and three-tier TP-monitor configurations
- Consistent support for local and global transactions, including support for the XA interface’s TM\_JOIN operation
- Improved scalability by providing the ability to concentrate connections, processes, and sessions across users on connections and 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

**OCI Object Support**

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

**Client-Side Object Cache**

The object cache is a client-side memory buffer that provides lookup and memory management support for objects. It stores and tracks object instances that have been fetched by an OCI application from the server to the client side. The object cache is created when the OCI environment is initialized. 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
**Associative and Navigational Interfaces**

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 `REFS`

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 `REFS`. This navigational interface provides functions for:

- Instantiating a copy of a referenceable persistent object (that is, of a persistent object with object ID in the client-side cache) by pinning its smart pointer or `REF`
- Traversing a sequence of objects that are connected to each other by traversing the `REFS` that point from one to the other
- Dynamically getting and setting values of an object's attributes

**OCI Runtime Environment for Objects**

OCI provides functions for objects 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

**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.

**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.

**OCI Support for Oracle Streams Advanced Queuing**

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

See Also: "OCI and Streams Advanced Queuing" on page 9-47

**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" on page D-3 for more information about Windows and XA applications
- Oracle Database Advanced Application Developer's Guide for information about developing applications with Oracle XA
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.

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 10g 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 in note 207303.1. See the website at

  [https://support.oracle.com/](https://support.oracle.com/)

Simplified Upgrading of Existing OCI Release 7 Applications

OCI has been significantly improved with many features since OCI release 7. Applications written to work with OCI release 7 have a smooth migration path to the current OCI release because of the interoperability of OCI release 7 clients with the current release of the Oracle Database, and of clients of the current release with Oracle Database release 7.

Specifically:

- Applications that use the OCI release 7.3 API work unchanged against the current release of Oracle Database. They do not need to be linked with the current client library.
OCI release 7 and the OCI calls of this release can be mixed in the same application and in the same transaction provided they are not mixed within the same statement execution.

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

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

- 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, 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.

### 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 Database continues to support these release 7.3 functions. Many of the earlier 7.x calls are available, but Oracle strongly recommends that new applications use the new calls to improve performance and provide increased functionality.

Table 1–1 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*. These 7.x calls are obsolete, meaning that OCI has replaced them with newer calls. Although the obsolete calls are now supported, they may not be supported in all future versions of OCI.

#### 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 Chapter 2, "OCI Programming Basics" of this guide for more information.

<table>
<thead>
<tr>
<th>7.x OCI Routine</th>
<th>Equivalent or Similar Later OCI Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>obindps(), obndra(), obndrn(), obndrv()</td>
<td>OCIBindByName(), OCIBindByPos() (Note: additional bind calls may be necessary for some data types)</td>
</tr>
<tr>
<td>obreak()</td>
<td>OCIBreak()</td>
</tr>
<tr>
<td>ocan()</td>
<td>none</td>
</tr>
<tr>
<td>oclose()</td>
<td>Note: cursors are not used in release 8.x or later</td>
</tr>
<tr>
<td>ocof(), ocon()</td>
<td>OCIStmExecute() with OCI_COMMIT_ON_SUCCESS mode</td>
</tr>
</tbody>
</table>
### Table 1–1 (Cont.) Obsolescent OCI Functions

<table>
<thead>
<tr>
<th>7.x OCI Routine</th>
<th>Equivalent or Similar Later OCI Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>ocom()</td>
<td>OCITransCommit()</td>
</tr>
<tr>
<td>odefin(), odefinps()</td>
<td>OCIDefineByPos() (Note: additional define calls may be necessary for some data types)</td>
</tr>
<tr>
<td>odescr()</td>
<td>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.</td>
</tr>
<tr>
<td>odesssp()</td>
<td>OCIDescribeAny()</td>
</tr>
<tr>
<td>oehms()</td>
<td>OCIErrorGet()</td>
</tr>
<tr>
<td>oexec(), oexn()</td>
<td>OCIStrmtExecute()</td>
</tr>
<tr>
<td>oexfet()</td>
<td>OCIStrmtExecute(), OCIStrmtFetch() (Note: result set rows can be implicitly prefetched)</td>
</tr>
<tr>
<td>ofen(), ofetch()</td>
<td>OCIStrmtFetch()</td>
</tr>
<tr>
<td>oflng()</td>
<td>none</td>
</tr>
<tr>
<td>ogetpi()</td>
<td>OCIStrmtGetPieceInfo()</td>
</tr>
<tr>
<td>olog()</td>
<td>OCILogon()</td>
</tr>
<tr>
<td>ologof()</td>
<td>OCILogoff()</td>
</tr>
<tr>
<td>onbclr(), onbset(), onbtst()</td>
<td>Note: nonblocking mode can be set or checked by calling OCIAttrSet() or OCIAttrGet() on the server context handle or service context handle</td>
</tr>
<tr>
<td>oopen()</td>
<td>Note: cursors are not used in release 8.x or later</td>
</tr>
<tr>
<td>oopt()</td>
<td>none</td>
</tr>
<tr>
<td>oparse()</td>
<td>OCIStrmtPrepare(); however, it is all local</td>
</tr>
<tr>
<td>opinit()</td>
<td>OCIEnvCreate()</td>
</tr>
<tr>
<td>oroll()</td>
<td>OCIStrmtRollback()</td>
</tr>
<tr>
<td>osetpi()</td>
<td>OCIStrmtSetPieceInfo()</td>
</tr>
<tr>
<td>sqlld2()</td>
<td>SQLSvcCtxGet or SQLEnvGet</td>
</tr>
<tr>
<td>sqllda()</td>
<td>SQLSvcCtxGet or SQLEnvGet</td>
</tr>
<tr>
<td>odsc()</td>
<td>Note: see odescr() preceding</td>
</tr>
<tr>
<td>oermsg()</td>
<td>OCIErrorGet()</td>
</tr>
<tr>
<td>olon()</td>
<td>OCILogon()</td>
</tr>
<tr>
<td>orlon()</td>
<td>OCILogon()</td>
</tr>
<tr>
<td>oname()</td>
<td>Note: see odescr() preceding</td>
</tr>
<tr>
<td>osql3()</td>
<td>Note: see oparse() preceding</td>
</tr>
</tbody>
</table>

**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 1–2.
Compatibility and Upgrading

Compatibility Between Different Releases of OCI and Servers

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

Existing 7.x applications with no new post-release 7.x calls have to be relinked with the current client-side library.

The application cannot use the object features of Oracle8i or later, and cannot get any of the performance or scalability benefits provided by those OCI releases.

Upgrading OCI

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

- Completely rewrite the application to use only current OCI calls (recommended).
- Incorporate current OCI post-release 7.x calls into the application, while still using 7.x calls for some operations.

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

Adding Post-Release 7.x OCI Calls to 7.x Applications

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

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

  See Also: See the description of "OCIServerAttach()" on page 16-27 and the description of "OCISessionBegin()" on page 16-30 for information about the logon calls necessary for applications that are maintaining multiple sessions

- After the server context handle has been initialized, it can be used with OCI post-release 7.x calls.
- To use release 7 OCI calls, convert the server context handle to an Lda_Def using OCISvcCtxToLda(), and pass the resulting Lda_Def to the 7.x calls.

<table>
<thead>
<tr>
<th>Table 1–2 OCI Functions Not Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI Routine</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>obind()</td>
</tr>
<tr>
<td>obindn()</td>
</tr>
<tr>
<td>odfinn()</td>
</tr>
<tr>
<td>odsrbn()</td>
</tr>
<tr>
<td>ologon()</td>
</tr>
<tr>
<td>osql()</td>
</tr>
</tbody>
</table>
To begin using post-release 7.x OCI calls again, the application must convert the Lda_Def back to a server context handle using OCILdaToSvcCtx().

The application may toggle between the Lda_Def and server context as often as necessary in the application. This approach allows an application to use a single connection, but two different APIs, to accomplish different tasks.

You can mix OCI 7.x and post-release 7.x calls within a transaction, but not within a statement. This lets you execute one SQL or PL/SQL statement with OCI 7.x calls and the next SQL or PL/SQL statement within that transaction with post-release 7.x OCI calls.

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

### OCI Instant Client

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

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

#### Table 1–3 OCI Instant Client Shared Libraries

<table>
<thead>
<tr>
<th>Linux and UNIX</th>
<th>Description for Linux and UNIX</th>
<th>Microsoft Windows</th>
<th>Description for Microsoft Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>libclntsh.so.11.1</td>
<td>Client Code Library oci.dll</td>
<td></td>
<td>Forwarding functions that applications link with</td>
</tr>
<tr>
<td>libociei.so</td>
<td>OCI Instant Client Data Shared Library oraociei11.dll</td>
<td></td>
<td>Data and code</td>
</tr>
</tbody>
</table>
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 Windows.

See Also: "Fault Diagnosability in OCI" on page 10-32

Oracle Database 11g Release 1 library names are used in the table.

To use the Microsoft ODBC and OLEDB driver, you must copy ociw32.dll from the ORACLE_HOME\bin directory.

Benefits of Instant Client

Why use Instant Client?

- Installation involves copying a small number of files.
- 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 in Instant Client mode.
- It is simple for independent software vendors to package applications.

OCI Instant Client Installation Process

The Instant Client libraries can be installed by either choosing the Instant Client option from Oracle Universal Installer or by downloading and installing the Instant Client libraries from the OCI page (see the bottom of OCI page for the Instant Client link) on the Oracle Technology Network website:


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

1. Download and install the Instant Client shared libraries to an empty directory, such as instantclient_11_2, for Oracle Database release 11.2. Choose the Basic package.

2. Set the operating system shared library path environment variable to the directory from Step 1. For example, on Linux or UNIX, set LD_LIBRARY_PATH to instantclient_11_2. On Windows, set PATH to the instantclient_11_2 directory.

3. 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.

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

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

1. Invoke the Oracle Universal Installer and select the Instant Client option.

2. Install the Instant Client shared libraries to an empty directory, such as \instantclient_11_2, for release 11.2.

3. Set the LD_LIBRARY_PATH to the instant client directory to operate in instant client mode.

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.

If you did a complete client installation (by choosing the Admin option in Oracle Universal Installer), the Instant Client shared libraries are also installed. The locations of the Instant Client shared libraries in a full client installation are:

On Linux or UNIX:

libociei.so is in $ORACLE_HOME/instantclient
libclntsh.so.11.1 and libnnz11.so are in $ORACLE_HOME/lib

On Windows:

oraociei11.dll library is in ORACLE_HOME\instantclient
oci.dll, ociw32.dll, and orannzsbb11.dll are in ORACLE_HOME\bin

To enable running the OCI application in Instant Client mode, 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.11.1 on Linux or UNIX and oraocci11.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 libocijdbc11.so on Linux or UNIX and ocijdbc11.dll on Windows). Place all libraries in the Instant Client directory.
When to Use Instant Client

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

For development you can also use the Instant Client SDK.

See Also:
- "SDK for Instant Client" on page 1-27
- "Fault Diagnosability in OCI" on page 10-32

Patching 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 to patch the Instant Client shared libraries. Use the opatch utility to patch the Instant Client shared libraries.

After you apply the patch in an Oracle home environment, copy the files listed in Table 1–3 to the instant client directory, as described in "OCI Instant Client Installation Process" on page 1-17.

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

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

genzi -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.

Regeneration of Data Shared Library and Zip and RPM Files

The process to regenerate the data shared library and the zip and RPM files has changed for release 11.2 and later. Separate targets are added to create the data shared libraries, zip, and RPM files either individually or all at once. In previous releases, one target, ilibociei, was provided to build the data shared libraries, zip, and RPM files.
Now libociei 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. Therefore, separate targets have been added to regenerate the data shared libraries.

**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.

### 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.

### 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.

### 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.

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

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

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
```

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
```

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
```

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
```

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.

Database Connection Strings for OCI Instant Client
OCI Instant Client can make remote database connections in all the ways that ordinary SQL clients can. However, because Instant Client does not have the Oracle home environment and directory structure, some database naming methods require additional configuration steps.
All Oracle Net naming methods that do not require use of ORACLE_HOME or TNS_ADMIN (to locate configuration files such as tnsnames.ora or sqlnet.ora) work in the Instant Client mode. In particular, the connect_identifier in the OCIServerAttach() call can be specified in the following formats:

- A SQL Connect URL string of the form:
  
  \[\[host[:port]//service name\]\n
  For example:

  //dlsun242:5521/bjava21

- As an Oracle Net connect descriptor. For example:

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

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

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

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

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

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

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

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

**Examples of Instant Client Connect Identifiers**

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

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

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

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

The SQL*Plus connect identifier is:
The connect identifier can also be specified as:

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

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

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

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

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

Then you can invoke SQL*Plus with an empty connect identifier (you are prompted for the password):

```bash
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:

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

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

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

Then you can use the connect identifier `conn_str` for invoking SQL*Plus, or for your OCI connection.

---

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

---

If the 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 SQL*Plus can be invoked as previously, with the identifier `conn_str`:

```bash
setenv 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 SQL*Plus without a connect identifier:

```bash
setenv TWO_TASK conn_str
```

---

**Environment Variables for OCI Instant Client**

The `ORACLE_HOME` environment variable no longer determines the location of NLS, CORE, and error message files. An OCI-only application should not require `ORACLE_HOME` to be set. However, if it is set, it does not 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.

In the Instant Client mode, if the ORA_TZFILE variable is not set, then the larger, default, timezone_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.

On Linux or UNIX:

setenv 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.

To determine the versions of small and large timezone 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 entries 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.

---

**Instant Client Light (English)**

The Instant Client Light (English) 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.
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

Operation of Instant Client Light

OCI applications, by default, look for the OCI Data Shared Library, libociei.so (or Oraociei11.dll on Windows) on the LD_LIBRARY_PATH (PATH on Windows) to determine if the application should operate in the Instant Client mode. If this library is not found, then OCI tries to load the Instant Client Light Data Shared Library (see Table 1–4), libociicus.so (or Oraociicus11.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>
<thead>
<tr>
<th>Description for Linux and UNIX</th>
<th>Description for Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Code Library</td>
<td>ocli.dll</td>
</tr>
<tr>
<td>Forwarding functions that applications link with</td>
<td></td>
</tr>
</tbody>
</table>
Installation of Instant Client Light

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

- **From OTN**

  Go to the Instant Client link from the OCI URL (see the bottom of OCI page for the Instant Client link) on the Oracle Technology Network website:

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

  For Instant Client Light, download and unzip the basiclite.zip package in to an empty instantclient_11_2 directory.

- **From Client Admin Install**

  From the ORACLE_HOME/instantclient/light subdirectory, copy libociicus.so (or Oraociicus11.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 (Oraociicus11.dll on Windows), instead of the larger OCI Instant Client Data Shared Library, libociei.so (Oraociei11.dll on Windows).

- **From Oracle Universal Installer**

  When you select the Instant Client option from the Oracle Universal Installer, libociie.so (or Oraociei11.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 Oraociicus11.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 (Oraociei11.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_11_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_11_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.
SDK for Instant Client

The SDK can be downloaded from the Instant Client link on the OCI URL (see the bottom of OCI page for the Instant Client link) on the Oracle Technology Network website:

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

- 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_11_2 directory must be on the LD_LIBRARY_PATH before linking the application. The OCI and OCCI programs require the presence of libclntsh.so and libocci.so symbolic links in the instantclient_11_2 directory. demo.mk creates these before the link step. These symbolic links can also be created in a shell:

  ```
  cd instantclient_11_2
  ln -s libclntsh.so.11.1 libclntsh.so
  ln -s libocci.so.11.1 libocci.so
  ```

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

- Header File and Makefile Locations
- Overview of OCI Program Programming
- OCI Data Structures
- OCI Programming Steps
- Error Handling in OCI
- Additional Coding Guidelines
- 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

**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.

All demonstration programs and their related header files continue to reside in the `$ORACLE_HOME/rdbms/demo` directory. These demonstration files are installable only from the Examples media. See Appendix B 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
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 2–1 illustrates the flow of steps in an OCI application. "OCI Programming Steps" on page 2-12 describes each step in more detail.

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:** Chapter 11 through Chapter 15 for information about accessing and manipulating objects

## OCI Data Structures

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

### 7.x Upgrade Note:
Programmers who have previously written 7.x OCI applications 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. Descriptors are discussed in "OCI Descriptors" on page 2-9.

### 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.

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

*Table 2–1* 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>
<thead>
<tr>
<th>Description</th>
<th>C Data Type</th>
<th>Handle Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI environment handle</td>
<td>OCIEnv</td>
<td>OCI_HTYPE_ENV</td>
</tr>
<tr>
<td>OCI error handle</td>
<td>OCIError</td>
<td>OCI_HTYPE_ERROR</td>
</tr>
<tr>
<td>OCI service context handle</td>
<td>OCISvcCtx</td>
<td>OCI_HTYPE_SVCCTX</td>
</tr>
<tr>
<td>OCI statement handle</td>
<td>OCISstmt</td>
<td>OCI_HTYPE_STMT</td>
</tr>
<tr>
<td>OCI bind handle</td>
<td>OCIBind</td>
<td>OCI_HTYPE_BIND</td>
</tr>
<tr>
<td>OCI define handle</td>
<td>OCIDefine</td>
<td>OCI_HTYPE_DEFINE</td>
</tr>
</tbody>
</table>
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.

---

Table 2–1 (Cont.) OCI Handle Types

<table>
<thead>
<tr>
<th>Description</th>
<th>C Data Type</th>
<th>Handle Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI describe handle</td>
<td>OCIDescribe</td>
<td>OCI_HTYPE_DESCRIBE</td>
</tr>
<tr>
<td>OCI server handle</td>
<td>OCIServer</td>
<td>OCI_HTYPE_SERVER</td>
</tr>
<tr>
<td>OCI user session handle</td>
<td>OCISession</td>
<td>OCI_HTYPE_SESSION</td>
</tr>
<tr>
<td>OCI authentication information handle</td>
<td>OCIAuthInfo</td>
<td>OCI_HTYPE_AUTHINFO</td>
</tr>
<tr>
<td>OCI connection pool handle</td>
<td>OCICPool</td>
<td>OCI_HTYPE_CPOOL</td>
</tr>
<tr>
<td>OCI session pool handle</td>
<td>OCISPool</td>
<td>OCI_HTYPE_SPOOL</td>
</tr>
<tr>
<td>OCI transaction handle</td>
<td>OCITrans</td>
<td>OCI_HTYPE_TRANS</td>
</tr>
<tr>
<td>OCI complex object retrieval (COR) handle</td>
<td>OCIComplexObject</td>
<td>OCI_HTYPE_COMPLEXOBJECT</td>
</tr>
<tr>
<td>OCI thread handle</td>
<td>OCIThreadHandle</td>
<td>Not applicable</td>
</tr>
<tr>
<td>OCI subscription handle</td>
<td>OCISubscription</td>
<td>OCI_HTYPE_SUBSCRIPTION</td>
</tr>
<tr>
<td>OCI direct path context handle</td>
<td>OCIDirPathCtx</td>
<td>OCI_HTYPE_DIRPATH_CTX</td>
</tr>
<tr>
<td>OCI direct path function context handle</td>
<td>OCIDirPathFuncCtx</td>
<td>OCI_HTYPE_DIRPATH_FN_CTX</td>
</tr>
<tr>
<td>OCI direct path column array handle</td>
<td>OCIDirPathColArray</td>
<td>OCI_HTYPE_DIRPATH_COLUMN_ARRAY</td>
</tr>
<tr>
<td>OCI direct path stream handle</td>
<td>OCIDirPathStream</td>
<td>OCI_HTYPE_DIRPATH_STREAM</td>
</tr>
<tr>
<td>OCI process handle</td>
<td>OCIProcess</td>
<td>OCI_HTYPE_PROC</td>
</tr>
<tr>
<td>OCI administration handle</td>
<td>OCIAdmin</td>
<td>OCI_HTYPE_ADMIN</td>
</tr>
<tr>
<td>OCI HA event handle</td>
<td>OCIEvent</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
Handles lessen the need for global variables. Handles also make error reporting easier. An error handle is used to return errors and diagnostic information.

See Also: The example programs listed in Appendix B for sample code demonstrating the allocation and use of OCI handles

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.

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.

See Also: "Implementing Thread Safety" on page 8-25

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 2–2.

Figure 2–2   Components of a Service Context

- A server handle identifies a connection to a database. It translates into a physical connection in a connection-oriented transport mechanism.
- A user session handle defines a user’s roles and privileges (also known as the user’s security domain), and the operational context in which the calls execute.
A *transaction handle* defines the transaction in which the SQL operations are performed. The transaction context includes user session state information, including any fetch state and package instantiation.

Breaking the service context 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 `OCIAtrrSet()` 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" on page 8-1
- "OCI Environment Initialization" on page 2-13, and "Password and Session Management" on page 8-7 for more information about establishing a server connection and user session.

**Statement, Bind, and Define Handles**

*A statement handle* is the context that identifies a SQL or PL/SQL statement and its associated attributes, as shown in Figure 2–3.

![Statement Handles Diagram](image)

Information about input and output bind variables is stored in *bind handles*. The OCI library allocates a bind handle for each placeholder bound with the `OCIBindByName()` or `OCIBindByPos()` function. The user 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()`. 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 Chapter 5. 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" on page 5-7
- "Advanced Define Operations in OCI" on page 5-15

**Describe Handle**
The **describe handle** is used by the OCI describe call, OCIDescribeAny(). This call obtains information about schema objects in a database (for example, functions 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:** Chapter 6 for more information about using the OCIDescribeAny() function

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

**See Also:** "Complex Object Retrieval" on page 11-15

**Thread Handle**
For information about the thread handle, which is used in multithreaded applications, see "OCIThread Package" on page 8-26.

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

**See Also:** "Publish-Subscribe Notification in OCI" on page 9-54

**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 2–4 shows the different kinds of direct path handles.
Figure 2–4  Direct Path Handles

See Also:
- "Direct Path Loading Overview" on page 13-1
- "Direct Path Loading Handle Attributes" on page A-62

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

See Also:  "Connection Pooling in OCI" on page 9-1

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 2–1 set the user name in the session handle by writing to the OCI_ATTR_USERNAME attribute:

Example 2–1  Using the OCI_ATTR_USERNAME Attribute to Set the User Name in the Session Handle

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

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

Other OCI functions provide useful return data in handle attributes after the function completes. For example, when OCISqlExecutef() 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 2–2.

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

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

OCI descriptors and locators are opaque data structures that maintain data-specific information. Table 2–2 lists them, 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. See also the functions "OCIArrayDescriptorAlloc()" on page 16-48 and "OCIArrayDescriptorFree()" on page 16-50.

See Also:

- The description of "OCIArrayDescriptorAlloc()" on page 16-48 for an example showing how to allocate a large number of descriptors
- Appendix A, “Handle and Descriptor Attributes”

### Table 2–2 Descriptor Types

<table>
<thead>
<tr>
<th>Description</th>
<th>C Data Type</th>
<th>OCI Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snapshot descriptor</td>
<td>OCISnapshot</td>
<td>OCI_DTYPE_SNAP</td>
</tr>
<tr>
<td>Result set descriptor</td>
<td>OCIResult</td>
<td>OCI_DTYPE_RSET</td>
</tr>
<tr>
<td>LOB data type locator</td>
<td>OCILobLocator</td>
<td>OCI_DTYPE_LOB</td>
</tr>
<tr>
<td>BFILE data type locator</td>
<td>OCILobLocator</td>
<td>OCI_DTYPE_FILE</td>
</tr>
<tr>
<td>Read-only parameter descriptor</td>
<td>OCIParam</td>
<td>OCI_DTYPE_PARAM</td>
</tr>
<tr>
<td>ROWID descriptor</td>
<td>OCIRowid</td>
<td>OCI_DTYPE_ROWID</td>
</tr>
<tr>
<td>ANSI DATE descriptor</td>
<td>OCIDateTime</td>
<td>OCI_DTYPE_DATE</td>
</tr>
<tr>
<td>TIMESTAMP descriptor</td>
<td>OCIDateTime</td>
<td>OCI_DTYPE_TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE descriptor</td>
<td>OCIDateTime</td>
<td>OCI_DTYPE_TIMESTAMP_TZ</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH descriptor</td>
<td>OCIInterval</td>
<td>OCI_DTYPE_INTERVAL_YM</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND descriptor</td>
<td>OCIInterval</td>
<td>OCI_DTYPE_INTERVAL_DS</td>
</tr>
<tr>
<td>User callback descriptor</td>
<td>OCIUcb</td>
<td>OCI_DTYPE_UCB</td>
</tr>
<tr>
<td>Distinguished names of the database servers in a registration request</td>
<td>OCIServerDNs</td>
<td>OCI_DTYPE_SRVDN</td>
</tr>
<tr>
<td>Complex object descriptor</td>
<td>OCIComplexObjectComp</td>
<td>OCI_DTYPE_COMPLEXOBJECTCOMP</td>
</tr>
<tr>
<td>Advanced queuing enqueue options</td>
<td>OCIAQEnqOptions</td>
<td>OCI_DTYPE_AQENQ_OPTIONS</td>
</tr>
<tr>
<td>Advanced queuing dequeue options</td>
<td>OCIAQDeqOptions</td>
<td>OCI_DTYPE_AQDEQ_OPTIONS</td>
</tr>
<tr>
<td>Advanced queuing message properties</td>
<td>OCIAQMsgProperties</td>
<td>OCI_DTYPE_AQMSG_PROPERTIES</td>
</tr>
<tr>
<td>Advanced queuing agent</td>
<td>OCIQAQAgent</td>
<td>OCI_DTYPE_AQAGENT</td>
</tr>
<tr>
<td>Advanced queuing notification</td>
<td>OCIAQNotify</td>
<td>OCI_DTYPE_AQNFY</td>
</tr>
<tr>
<td>Advanced queuing listen options</td>
<td>OCIAQListenOpts</td>
<td>OCI_DTYPE_AQLIS_OPTIONS</td>
</tr>
<tr>
<td>Advanced queuing message properties</td>
<td>OCIAQListenOpts</td>
<td>OCI_DTYPE_AQLIS_MSG_PROPERTIES</td>
</tr>
<tr>
<td>Change notification</td>
<td>None</td>
<td>OCI_DTYPE_CHDES</td>
</tr>
<tr>
<td>Table change</td>
<td>None</td>
<td>OCI_DTYPE_TABLE_CHDES</td>
</tr>
<tr>
<td>Row change</td>
<td>None</td>
<td>OCI_DTYPE_ROW_CHDES</td>
</tr>
</tbody>
</table>
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" on page 2-10 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

**Snapshot Descriptor**

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

**See Also:** "Execution Snapshots" on page 4-6 for more information about OCIStmtExecute() and database snapshots

**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.

**See Also:**
- "Binding LOB Data" on page 5-9
- "Defining LOB Data" on page 5-16

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 BLOBs or CLOBs, and `OCI_DTYPE_FILE` for BFILES.

---

**Caution:** The two LOB locator types are not interchangeable. When binding or defining a BLOB or CLOB, the application must take care that the locator is properly allocated 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.

**See Also:**
- Chapter 7, “LOB and BFILE Operations”
- "Binding LOB Data” on page 5-9
- "Defining LOB Data” on page 5-16

---

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

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

**See Also:** Chapter 6 and "Describing Select-List Items” on page 4-9 for more information about obtaining and using parameter descriptors

---

**ROWID Descriptor** The ROWID descriptor, `OCIRowid`, is used by applications that 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.

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

---

**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.
See Also:
- Chapter 3 for more information about these data types
- Chapter 19 for descriptions of the functions that operate on these data types

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

See Also:  "Complex Object Retrieval" on page 11-15 for information about the complex object descriptor and its use

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

See Also:
- "OCI and Streams Advanced Queuing" on page 9-47
- "Publish-Subscribe Registration Functions in OCI" on page 9-55

User Memory Allocation  The OCIDescriptorAlloc() call has an xtramem_sz parameter in its parameter list. This 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.

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.
See Also:

- The first sample program in Appendix B for an example showing the use of OCI calls for processing SQL statements.
- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for a detailed description of the special case of dynamically providing data at run time.
- "Binding and Defining Arrays of Structures in OCI" on page 5-18 for a description of the special considerations for operations involving arrays of structures.
- "Error Handling in OCI" on page 2-20 for an outline of the steps involved in processing a SQL statement within an OCI program.
- "Overview of OCI Multithreaded Development" on page 8-24 for information about using the OCI to write multithreaded applications.
- "SQL Statements" on page 1-4 for more information about types of SQL statements.

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

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

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

**OCI Environment Initialization**

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

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

- "Creating the OCI Environment" on page 2-13
- "Allocating Handles and Descriptors" on page 2-14
- "Application Initialization, Connection, and Session Creation" on page 2-14

**Creating the OCI Environment**

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

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

- Run in a threaded environment (`mode = OCI_THREAD`).
- 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()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated) for more information about the initialization calls
- "Overview of OCI Multithreaded Development" on page 8-24
- Chapter 11, Chapter 12, Chapter 13, Chapter 14, and Chapter 15
- "Publish-Subscribe Notification in OCI" on page 9-54

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(), allocates the handles for you.

You can allocate the types of handles listed in Table 2–1 with OCIHandleAlloc(). Depending on the functionality of your application, it must allocate some or all of these handles.

Application Initialization, Connection, and Session Creation

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

Following this step, the application has several options for establishing 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 OCEnvInit() calls. OCIInitialize() and OCEnvInit() calls are supported for backward compatibility.

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 2–3 shows what a call to `OCILogon2()` looks like for a single user session with user name `hr`, password `hr`, and database `oracledb`.

**Example 2–3  Using the OCILogon2 Call for a Single User Session**

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

The parameters to this call include the service context handle (which has been initialized), the user name, the user’s password, and the name of the database that are used to establish the connection. With the last parameter, `mode`, set to `OCI_DEFAULT`, this call has the same effect as calling the older `OCILogon()`. Use `OCILogon2()` for any new applications. The server and user session handles are implicitly allocated by this function.

If an application uses this logon method, the service context, server, and user session handles 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.

---

For information regarding operating systems providing facilities for spawning processes that allow child processes to reuse state created by their parent process, see "Operating System Considerations" on page 2-22. This section explains why the child process must not use the same database connection as created by the parent.

**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 11g, 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.

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 2–4 through Example 2–11 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 2–4, 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 2–4.

**Example 2–4 Enabling a Local User to Serve as a Proxy for Another User**

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

When user name dilbert is acting on behalf of joe, use the connection string shown in Example 2–5. (The user name dilbert has the password tiger123).

**Example 2–5 Connection String to Use for the Proxy User**

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

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

In Example 2–6, "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 2–6.

**Example 2–6 Preserving Case Sensitivity When Enabling a Local User to Serve as a Proxy for Another User**

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

When "Dilbert" is acting on behalf of "Joe", use the connection string shown in Example 2–7. Be sure to include the double quotation marks ("" characters.

**Example 2–7 Preserving Case Sensitivity in the Connection String**

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

When the proxy user is created as "dilbert[mybert]", use the connection string shown in Example 2–8 to connect to the database. (The left and right brackets "[" and "]" are entered in the connection string.)

**Example 2–8 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;
In Example 2-9, 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 2-9 shows the connect statement to use.

**Example 2-9  Using "dilbert[mybert]"["joe[myjoe]"] in the Connection String**
*dilbert[mybert]"["joe[myjoe]"*/tiger123

In Example 2-10, you can set the target user name by using the ALTER USER statement.

**Example 2-10  Setting the Target User Name**
ALTER USER joe GRANT CONNECT THROUGH dilbert;

Then, as shown in Example 2-11, 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 2-11  Using OCI to Set the OCI_ATTR_PROXY_CLIENT Attribute and the Proxy dilbert**

OCIAttrSet(session, OCI_HTYPE_SESSION, (void *)&"dilbert",
(void*)strlen("dilbert"), OCI_ATTR_USERNAME,
error_handle);

OCIAttrSet(session, OCI_HTYPE_SESSION, (void *)&"tiger123",
(void*)strlen("tiger123"), OCI_ATTR_PASSWORD,
error_handle);

OCIAttrSet(session, OCI_HTYPE_SESSION, (void *)&"joe",
(void*)strlen("joe"), OCI_ATTR_PROXY_CLIENT,
error_handle);

**See Also:**
- "OCI_ATTR_PROXY_CLIENT" on page A-20
- Oracle Database Security Guide for a discussion of proxy authentication
- "Password and Session Management" on page 8-7
- "OCIAttrSet()" on page 16-53

**Caution:** 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.

**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.
OCI Programming Steps

- **OCIEnvCreate()** - 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 **OCIEnvCreate()** 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 **OCIEnvCreate()** call must be made with the same service context handle and a new session context handle.

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

**See Also:**
- "Connect, Authorize, and Initialize Functions" on page 16-3
- Chapter 9 for more information about maintaining multiple sessions, transactions, and connections
- "Client Character Set Control from OCI" on page 2-30 for the use of **OCIEnvNlsCreate()**

**Example 2–12** 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.

**Example 2–12  Creating and Initializing an OCI Environment**

```c
#include <oci.h>
...
main()
{
...
OCIEnv  *myenvhp;    /* the environment handle */
OCIServer *mysrvhp;    /* the server 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) OCIServerAttach (mysrvhp, myusrhp, OCI_DEFAULT);

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

/* allocate a service handle */
(void) OCIHandleAlloc ((void *)myenvhp, (void **)mysvchp,
```

---

**Note:**
- **OCIEnvCreate** sets up an environment for executing SQL and PL/SQL statements.
- **OCIEnvAttach** is used to attach to an existing environment.
- **OCIEnvNlsCreate** is used to create a multilingual environment.
- **OCIEnvCreate** can be called multiple times to create a new environment.
- **OCIEnvAttach** is used to attach to an existing environment.
- **OCIEnvCreate** sets the mode to be the threaded and object environment.

---

**See Also:**
- "Connect, Authorize, and Initialize Functions" on page 16-3
- Chapter 9 for more information about maintaining multiple sessions, transactions, and connections
- "Client Character Set Control from OCI" on page 2-30 for the use of **OCIEnvNlsCreate()**
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);

...

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

**Processing SQL Statements in OCI**

Chapter 4 outlines the specific steps involved in processing SQL statements in OCI.

**Commit or Roll Back Operations**

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

**Note:** By using the OCI_COMMIT_ON_SUCCESS mode of the
OCISqlExecut() 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" on page 2-5
- "OCI Support for Transactions" on page 8-1

Terminating the Application
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 Appendix B 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.

Error Handling in OCI
OCI function calls have a set of return codes, listed in Table 2-3, 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" on page 2-22 for exceptions
- "OCIErrorGet()" on page 17-165 for complete details and an example of usage
- "Server Handle Attributes" on page A-11
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.

## Return and Error Codes for Data

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

### See Also:
"Indicator Variables" on page 2-24
For truncated data, `data_len` is the actual length of the data that has been truncated if this length is less than or equal to `SB2MAXVAL`. Otherwise, the indicator is set to `-2`.

### Functions Returning Other Values

Some functions return values other than the OCI error codes listed in Table 2–3. When 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.

### Additional Coding Guidelines

This section explains some additional issues when coding OCI applications.

### 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.

**See Also:**

- "Overview of OCI Multithreaded Development" on page 8-24
  - "OCIThread Package" on page 8-26

For better performance with many concurrently opened connections, consider pooling them.

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

<table>
<thead>
<tr>
<th>State of Data</th>
<th>Return Code</th>
<th>Indicator - Not provided</th>
<th>Indicator - Provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null data</td>
<td>OCI_ERROR</td>
<td>OCI_SUCCESS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Error = 1405</td>
<td>Error = 0</td>
<td>Indicator = -1</td>
</tr>
<tr>
<td></td>
<td>Return code = 1405</td>
<td>Return code = 1405</td>
<td></td>
</tr>
<tr>
<td>Truncated data</td>
<td>OCI_ERROR</td>
<td>OCI_ERROR</td>
<td></td>
</tr>
<tr>
<td>Not provided</td>
<td>Error = 1406</td>
<td>Error = 1406</td>
<td>Indicator = data_len</td>
</tr>
<tr>
<td>Truncated data</td>
<td>OCI_SUCCESS_WITH_INFO</td>
<td>OCI_SUCCESS_WITH_INFO</td>
<td></td>
</tr>
<tr>
<td>Provided</td>
<td>Error = 24345</td>
<td>Error = 24345</td>
<td>Indicator = data_len</td>
</tr>
<tr>
<td></td>
<td>Return code = 1405</td>
<td>Return code = 1405</td>
<td></td>
</tr>
</tbody>
</table>
Parameter Types

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

See Also:  "Connect, Authorize, and Initialize Functions" on page 16-3 for more information about parameter data types and parameter passing conventions

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.

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.

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.

Inserting Nulls into a Column

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

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

\begin{verbatim}
INSERT INTO empl (ename, empno, deptno)
VALUES (NULL, 8010, 20)
\end{verbatim}

- Use indicator variables in the OCI bind call. See "Indicator Variables" on page 2-24.

- Insert a \texttt{NULL} to set both the buffer length and maximum length parameters to zero on a bind call.

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

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.
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.

Input

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

<table>
<thead>
<tr>
<th>Input Indicator Value</th>
<th>Action Taken by Oracle Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Oracle Database assigns a NULL to the column, ignoring the value of the input variable.</td>
</tr>
<tr>
<td>&gt;=0</td>
<td>Oracle Database assigns the value of the input variable to the column.</td>
</tr>
</tbody>
</table>

Output

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

<table>
<thead>
<tr>
<th>Output Indicator Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>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.</td>
</tr>
<tr>
<td>-1</td>
<td>The selected value is null, and the value of the output variable is unchanged.</td>
</tr>
<tr>
<td>0</td>
<td>Oracle Database assigned an intact value to the host variable.</td>
</tr>
<tr>
<td>&gt;0</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
Canceling 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" on page 8-24
- "OCIThread Package" on page 8-26

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 3–2) when binding rowid to :2.

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

See Also: ■ ”UROWID” on page 3-5 and ”DATE” on page 3-13 for more information about ROWIDs

■ ”External Data Types” on page 3-6 for a table of external data types and codes

Reserved Words

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

See Also: 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

Oracle Reserved Namespaces

Table 2–5 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 2–5 Oracle Reserved Namespaces

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>XA</td>
<td>External functions for XA applications only</td>
</tr>
<tr>
<td>SQ</td>
<td>External SQLLIB functions used by Oracle Precompiler and SQL*Module applications</td>
</tr>
<tr>
<td>0, OCI</td>
<td>External OCI functions internal OCI functions</td>
</tr>
<tr>
<td>UPI, KP</td>
<td>Function names from the Oracle UPI layer</td>
</tr>
</tbody>
</table>
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 OCIRestart() are exceptions to the rule. These calls are allowed so that the caller can stop an OCI call that has been started.

Nonblocking Mode in OCI

Note: Because nonblocking mode requires the caller to repeat the same call until it completes, it increases CPU usage. Instead, use multithreaded mode.
OCI provides the ability to establish a server connection in **blocking mode** or **nonblocking mode**. When a connection is made in blocking mode, an OCI call returns control to an OCI client application only when the call completes, either successfully or in error. With the nonblocking mode, control is immediately returned to the OCI program if the call could not complete, and the call returns a value of `OCI_STILL_EXECUTING`.

In nonblocking mode, an application must test the return code of each OCI function to see if it returns `OCI_STILL_EXECUTING`. If it does, the OCI client can continue to process program logic while waiting to retry the OCI call to the server. This mode is particularly useful in graphical user interface (GUI) applications, real-time applications, and in distributed environments.

The nonblocking mode is not interrupt-driven. Rather, it is based on a polling paradigm, which means that the client application 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()`

### Setting Blocking Modes

You can modify or check an application’s blocking status by calling `OCIAttrSet()` to set the status, or `OCIAttrGet()` to read the status on the server context handle with the `attrtype` parameter set to `OCI_ATTR_NONBLOCKING_MODE`. You must set this attribute only after `OCISessionBegin()` or `OCILogon2()` has been called. Otherwise, an error is returned.

**See Also:**  
- "Overview of OCI Multithreaded Development" on page 8-24  
- "OCIThread Package" on page 8-26  

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

---

2-28 Oracle Call Interface Programmer's Guide
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.

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 32512 bytes, because of the overhead of control structures.
OCI Globalization Support

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

Client Character Set Control from OCI

The function `OCIEnvNlsCreate()` enables you to set character set information in applications independently from NLS_LANG and NLS_NCHAR settings. One application can have several environment handles initialized within the same system environment using different client-side character set IDs and national character set IDs. For example:

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

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

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

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

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

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

See Also:  "OCIEnvNlsCreate()" on page 16-17  

- "Setting Client Character Sets in OCI" on page 5-27 for a pseudocode fragment that illustrates a sample usage of these calls
Character Control and OCI Interfaces

The OCINlsGetInfo() function returns information about 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 NULL 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() and OCIBindByPos() 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() 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.

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.

Character Set Support in OCI

See "Character-Length Semantics Support in Describe Operations" on page 6-17 and "Character Conversion in OCI Binding and Defining" on page 5-26 for a complete discussion of character set support in OCI.

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:
- Chapter 22, "OCI Globalization Support Functions"
- Oracle Database Globalization Support Guide for information about OCI programming with Unicode

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" on page 22-3
- "OCINlsEnvironmentVariableGet()" on page 22-6

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

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

The code in Example 2-13 retrieves locale information and checks for errors.

Example 2–13  Getting Locale Information in OCI

```c
sword MyPrintLinguisticName(envhp, errhp)
OCIEnv   *envhp;
OCIError *errhp;
{ 
```
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” on page 22-14

Example 2–14 shows a simple case of manipulating strings.

**Example 2–14 Basic String Manipulation in OCI**

```c
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);
}

The OCI character classification functions are described in detail in "OCI Character Classification Functions" on page 22-44.

Example 2–15 shows how to classify characters in OCI.

**Example 2–15  Classifying Characters in OCI**

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

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

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

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

  return(status);
}
```
Converting Character Sets in OCI

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

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

Example 2–16 shows a simple conversion into Unicode.

**See Also:** “OCI Character Set Conversion Functions” on page 22-57

**Example 2–16 Converting Character Sets in OCI**

```c
/* 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);

        /* size_t dstLen = 0; */
        /* size_t srcLen = 0; */
        /* OraText tb[OCI_NLS_MAXBUFSZ]; */
        /* ub2 cid; */

        /* get OCIEnv character set */
        checkerr(errhp, OCINlsGetInfo(envhp, errhp, tb, sizeofs(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 = OCIMultiByte_strlen(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);
    }

See Also: “OCI Character Set Conversion Functions” on page 22-57
```
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
- "OCI Messaging Functions" on page 22-63

Example 2–17 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 2–17 Retrieving a Message from a Text Message File

```c
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);
```

Imsgen Utility

The Imsgen 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 Imsgen utility is as follows:

```
Imsgen 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.

Guidelines for Text Message Files

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:
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)′
```

**An Example of Creating a Binary Message File from a Text Message File**

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

<table>
<thead>
<tr>
<th>Imsgen Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>$HOME/myApplication</td>
</tr>
<tr>
<td>facility</td>
<td>imp</td>
</tr>
<tr>
<td>language</td>
<td>AMERICAN</td>
</tr>
<tr>
<td>text_file</td>
<td>impus.msg</td>
</tr>
</tbody>
</table>

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
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

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 3–1).

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:

```sql
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()" 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 3–2 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. See the description of the DATE external data type on page 3-13 for more information.

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).

  **See Also:** "Internal Data Types" on page 3-3

- **External OCI data types**, which are used to specify host language representations of Oracle data.
Internal Data Types

See Also: "External Data Types" on page 3-6 and "Using External Data Type Codes" on page 3-3

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

See Also: "Typecodes" on page 3-25, and "Relationship Between SQLT and OCI_TYPECODE Values" on page 3-27

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" on page 3-3.

See Also:

- Oracle Database SQL Language Reference for detailed information about Oracle internal data types
- "Describing Select-List Items" on page 4-9 for information about describing select-list items in a query

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 3–2 for a complete list of the external data types and data type codes

Internal Data Types

Table 3–1 lists the internal Oracle Database data types (also known as *built-in*), along with each type's maximum internal length and data type code.

<table>
<thead>
<tr>
<th>Internal Oracle Database Data Type</th>
<th>Maximum Internal Length</th>
<th>Data Type Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2, NVARCHAR2</td>
<td>4000 bytes</td>
<td>1</td>
</tr>
<tr>
<td>NUMBER</td>
<td>21 bytes</td>
<td>2</td>
</tr>
<tr>
<td>LONG</td>
<td>2^31-1 bytes (2 gigabytes)</td>
<td>8</td>
</tr>
</tbody>
</table>
LONG, RAW, LONG RAW, VARCHAR2

You can use the piecewise capabilities provided by OCIBindByName(), OCIBindByPos(), OCIDefineByPos(), OCIStmtGetPieceInfo(), and OCIStmtSetPieceInfo() to perform inserts, updates or fetches involving column data of the LONG, RAW, LONG RAW, and VARCHAR2 data types.

Character Strings and Byte Arrays

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

CHAR, VARCHAR2, and LONG columns normally hold character data. RAW and LONG RAW hold bytes that are not interpreted as characters (for example, pixel values in a
bit-mapped 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.

**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 higher.

The following host variables can be bound to Universal ROWIDs:

- `SQLT_CHR (VARCHAR2)`
- `SQLT_VCS (VARCHAR)`
- `SQLT_STR (NULL-terminated string)`
- `SQLT_LVC (LONG VARCHAR)`
- `SQLT_AFC (CHAR)`
- `SQLT_AVC (CHAR2)`
- `SQLT_VST (OCI String)`
- `SQLT_RDD (ROWID descriptor)`

**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)
External Data Types

For best performance, use external types `SQLT_BFLOAT` and `SQLT_BDOUBLE` in conjunction with the `BINARY_FLOAT` and `BINARY_DOUBLE` data types.

External Data Types

Table 3–2 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>
<thead>
<tr>
<th>External Data Type</th>
<th>Code</th>
<th>Program Variable¹</th>
<th>OCI-Defined Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2</td>
<td>1</td>
<td>char[n]</td>
<td>SQLT_CHR</td>
</tr>
<tr>
<td>NUMBER</td>
<td>2</td>
<td>unsigned char[21]</td>
<td>SQLT_NUM</td>
</tr>
<tr>
<td>8-bit signed INTEGER</td>
<td>3</td>
<td>signed char</td>
<td>SQLT_INT</td>
</tr>
<tr>
<td>16-bit signed INTEGER</td>
<td>3</td>
<td>signed short, signed int</td>
<td>SQLT_INT</td>
</tr>
<tr>
<td>32-bit signed INTEGER</td>
<td>3</td>
<td>signed int, signed long</td>
<td>SQLT_INT</td>
</tr>
<tr>
<td>64-bit signed INTEGER</td>
<td>3</td>
<td>signed long, signed long</td>
<td>SQLT_INT</td>
</tr>
<tr>
<td>FLOAT</td>
<td>4</td>
<td>float, double</td>
<td>SQLT_FLT</td>
</tr>
<tr>
<td>NULL-terminated STRING</td>
<td>5</td>
<td>char[n+1]</td>
<td>SQLT_STR</td>
</tr>
<tr>
<td>VARNUM</td>
<td>6</td>
<td>char[22]</td>
<td>SQLT_VNU</td>
</tr>
<tr>
<td>LONG</td>
<td>8</td>
<td>char[n]</td>
<td>SQLT_LNG</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>9</td>
<td>char[n+sizeof(short integer)]</td>
<td>SQLT_VCS</td>
</tr>
<tr>
<td>DATE</td>
<td>12</td>
<td>char[7]</td>
<td>SQLT_DAT</td>
</tr>
<tr>
<td>VARRAW</td>
<td>15</td>
<td>unsigned char[n+sizeof(short integer)]</td>
<td>SQLT_VBI</td>
</tr>
<tr>
<td>native float</td>
<td>21</td>
<td>float</td>
<td>SQLT_BFLOAT</td>
</tr>
<tr>
<td>native double</td>
<td>22</td>
<td>double</td>
<td>SQLT_BDOUBLE</td>
</tr>
<tr>
<td>RAW</td>
<td>23</td>
<td>unsigned char[n]</td>
<td>SQLT_BIN</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>24</td>
<td>unsigned char[n]</td>
<td>SQLT_LBI</td>
</tr>
</tbody>
</table>
The following two types are internal to PL/SQL and cannot be returned as values by OCI:

- Boolean, SQLT_BOL
- Record, SQLT_REC

**VARCHAR2**

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

---

**Table 3–2 (Cont.) External Data Types and Codes**

<table>
<thead>
<tr>
<th>External Data Type</th>
<th>Code</th>
<th>Program Variable</th>
<th>OCI-Defined Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED INT</td>
<td>68</td>
<td>unsigned</td>
<td>SQLT_UIN</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>94</td>
<td>char[n+sizeof(integer)]</td>
<td>SQLT_LVC</td>
</tr>
<tr>
<td>LONG VARRAW</td>
<td>95</td>
<td>unsigned char[n+sizeof(integer)]</td>
<td>SQLT_LVB</td>
</tr>
<tr>
<td>CHAR</td>
<td>96</td>
<td>char[n]</td>
<td>SQLT_AFC</td>
</tr>
<tr>
<td>CHARZ</td>
<td>97</td>
<td>char[n+1]</td>
<td>SQLT_AVC</td>
</tr>
<tr>
<td>ROWID descriptor</td>
<td>104</td>
<td>OCIRowid *</td>
<td>SQLT_RDD</td>
</tr>
<tr>
<td>NAMED DATATYPE</td>
<td>108</td>
<td>struct</td>
<td>SQLT_NTY</td>
</tr>
<tr>
<td>REF</td>
<td>110</td>
<td>OCIRef</td>
<td>SQLT_REF</td>
</tr>
<tr>
<td>Character LOB descriptor</td>
<td>112</td>
<td>OCIlobLocator2</td>
<td>SQLT_CLOB</td>
</tr>
<tr>
<td>Binary LOB descriptor</td>
<td>113</td>
<td>OCIlobLocator2</td>
<td>SQLT_BLOB</td>
</tr>
<tr>
<td>Binary FILE descriptor</td>
<td>114</td>
<td>OCIlobLocator</td>
<td>SQLT_FILE</td>
</tr>
<tr>
<td>OCI STRING type</td>
<td>155</td>
<td>OCIString</td>
<td>SQLT_VST3</td>
</tr>
<tr>
<td>OCI DATE type</td>
<td>156</td>
<td>OCIDate *</td>
<td>SQLT_ODT3</td>
</tr>
<tr>
<td>ANSI DATE descriptor</td>
<td>184</td>
<td>OCIDateTime *</td>
<td>SQLT_DATE</td>
</tr>
<tr>
<td>TIMESTAMP descriptor</td>
<td>187</td>
<td>OCIDateTime *</td>
<td>SQLT_TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>188</td>
<td>OCIDateTime *</td>
<td>SQLT_TIMESTAMP_TZ</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>189</td>
<td>OCIInterval *</td>
<td>SQLT_INTERVAL_YM</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>190</td>
<td>OCIInterval *</td>
<td>SQLT_INTERVAL_DS</td>
</tr>
<tr>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>232</td>
<td>OCIDateTime *</td>
<td>SQLT_TIMESTAMP_LTZ</td>
</tr>
</tbody>
</table>

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.

---

**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. See Chapter 12 for more information about this data type.
Input
The value_sz parameter determines the length in the OCIBindByName() or OCIBindByPos() call.

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.

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

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

To check if a NULL is returned or if character truncation has occurred, include an indicator parameter in the OCIDefineByPos() call. Oracle 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" on page 2-24

NUMBER

You should not need to use NUMBER as an external data type. If you do use it, 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.
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()` 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" on page 12-10
- "VARNUM" on page 3-12 for a description of the internal `NUMBER` format

### 64-Bit Integer Host Data Type

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()`, `OCIBindByName()`, and `OCIBindByPos()` 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.
Example 3–1  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, ...);
...

Example 3–2 shows a code fragment that illustrates binding 8-byte integer data types for OUT binds of a DML returning statement.

Example 3–2  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 emno = :2
    RETURNING sal INTO :out1";
...

/* Declare all handles to be used in program. */

OCIStmt    *stmthp;
OCIErr    *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 */

**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.

**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.

---

**STRING**

The `NULL`-terminated `STRING` format behaves like the `VARCHAR2` format, except that the string must contain a `NULL` terminator character. This data type is most useful for C language programs.

**Input**

The string length supplied in the `OCIBindByName()` or `OCIBindByPos()` call limits the scan for the `NULL` terminator. If the `NULL` terminator is not found within the length specified, Oracle 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.
External Data Types

---

**Note:** You cannot pass -1 for the string length parameter of a NULL-terminated string

---

**Output**

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

A NULL select-list item returns a NULL terminator character in the first character position. An ORA-01405 error is also possible.

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

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Length Byte</th>
<th>Exponent Byte</th>
<th>Mantissa Bytes</th>
<th>Terminator Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>128</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>193</td>
<td>6</td>
<td>Not applicable</td>
</tr>
<tr>
<td>-5</td>
<td>3</td>
<td>62</td>
<td>96</td>
<td>102</td>
</tr>
<tr>
<td>2767</td>
<td>3</td>
<td>194</td>
<td>28, 68</td>
<td>Not applicable</td>
</tr>
<tr>
<td>-2767</td>
<td>4</td>
<td>61</td>
<td>74, 34</td>
<td>102</td>
</tr>
<tr>
<td>100000</td>
<td>2</td>
<td>195</td>
<td>11</td>
<td>Not applicable</td>
</tr>
<tr>
<td>1234567</td>
<td>5</td>
<td>196</td>
<td>2, 24, 46, 68</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

**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.

**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.

**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 3–4.

<table>
<thead>
<tr>
<th>Byte</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Century</td>
</tr>
<tr>
<td>2</td>
<td>Year</td>
</tr>
<tr>
<td>3</td>
<td>Month</td>
</tr>
<tr>
<td>4</td>
<td>Day</td>
</tr>
<tr>
<td>5</td>
<td>Hour</td>
</tr>
<tr>
<td>6</td>
<td>Minute</td>
</tr>
<tr>
<td>7</td>
<td>Second</td>
</tr>
</tbody>
</table>

Table 3–4 Format of the DATE Data Type

<table>
<thead>
<tr>
<th>Byte</th>
<th>Century</th>
<th>Year</th>
<th>Month</th>
<th>Day</th>
<th>Hour</th>
<th>Minute</th>
<th>Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>119</td>
<td>192</td>
<td>11</td>
<td>30</td>
<td>16</td>
<td>18</td>
<td>1</td>
</tr>
</tbody>
</table>

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)" on page 12-5 for more information about the OCIDate data type
- "Datetime and Interval Data Type Descriptors" on page 3-19 for information about DATETIME and INTERVAL data types

**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.

**See Also:** Oracle Database SQL Language Reference
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 “Raw (OCIRaw)” on page 12-13 for more information about this data type.

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.

LONG RAW

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).

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.

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.

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.

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.

See Also: Oracle Database SQL Language Reference
Input
The length is determined by the value_sz parameter in the OCIBindByName() or OCIBindByPos() call.

---

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

---

If the value_sz parameter is zero, Oracle 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 CHARs.

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.

Output
Specify the desired length for the return value in the value_sz parameter of the OCIDeclareByPos() call. If zero is specified for the length, no data is returned.

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

To check whether a NULL is returned or character truncation occurs, include an indicator parameter or array of indicator parameters in the OCIDeclareByPos() call. An indicator parameter is set to -1 when a NULL is fetched and to the original column length when the returned value is truncated. Otherwise, it is set to zero. If you do not specify an indicator parameter and a NULL is selected, the fetch call returns an ORA-01405 error.

**See Also:** "Indicator Variables" on page 2-24

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.

**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.

---

**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" on page 12-20 for more information about working with named data types in OCI
- Chapter 15 for information about how named data types are represented as C structs

**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:** Chapter 14 for more information about working with `REF`s in the OCI

**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:

```sql
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 \texttt{UPDATE} statement. The \texttt{prefetch} operation fetches all \texttt{ROWIDS} on a \texttt{SELECT} for \texttt{UPDATE}; use \texttt{prefetching} and then a single row fetch.

You access rowids using a \texttt{ROWID} descriptor, which you can use as a bind or define variable.

\textbf{See Also:} "OCI Descriptors" on page 2-9 and "Positioned Updates and Deletes" on page 2-25 for more information about the use of the \texttt{ROWID} descriptor

\section*{LOB Descriptor}

A LOB (large object) stores binary or character data up to 128 terabytes (TB) in length. Binary data is stored in a \texttt{BLOB} (binary LOB), and character data is stored in a \texttt{CLOB} (character LOB) or \texttt{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 \textit{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 \texttt{OCILobLocator}.

\begin{quote}
\textbf{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 \texttt{CLOB} columns or \texttt{RAW} data for \texttt{BLOB} columns.
\end{quote}

\textbf{See Also:}
- "OCI Descriptors" on page 2-9 for more information about descriptors, including the LOB locator
- \textit{Oracle Database SQL Language Reference} and \textit{Oracle Database SecureFiles and Large Objects Developer's Guide} for more information about LOBs
- "Binding LOB Data" on page 5-9
- "Defining LOB Data" on page 5-16

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 \texttt{OCI Descriptor Alloc()} function.

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

\textbf{See Also:} Chapter 7 for more information about OCI LOB functions

The data type codes available for binding or defining LOBs are:
External Data Types

- **SQLT_BLOB** - A binary LOB data type
- **SQLT_CLOB** - A character LOB data type

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

- To write into or read from an **NCLOB**, the user must set the character set form (**csfrm**) parameter to be **SQLCS_NCHAR**.
- The amount (**amtp**) parameter in calls involving **CLOBs** and **NCLOBs** is always interpreted in terms of characters, rather than bytes, for fixed-width character sets.

**See Also:** "LOB and BF ile Functions in OCI" on page 7-8

**BFILE**

Oracle Database supports access to binary files (**BFILES**). 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 **BFILES** 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

**BLOB**

The **BLOB** data type stores unstructured binary large objects. **BLOBs** can be thought of as bit streams with no character set semantics. **BLOBs** can store up to 128 terabytes of binary data.

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

**CLOB**

The **CLOB** data type stores fixed-width or variable-width character data. **CLOBs** can store up to 128 terabytes of character data.

**CLOBs** 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.
**NCLOB**

An **NCLOB** is a national character version of a **CLOB**. It stores fixed-width, single-byte or multibyte national character set (**NCHAR**) data, or variable-width character set data. **NCLOBs** can store up to 128 terabytes of character text data.

**NCLOBs** have full transactional support; changes made through **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.

### Datetime and Interval Data Type Descriptors

The datetime and interval data type descriptors are briefly summarized here.

**See Also:** *Oracle Database SQL Language Reference*

#### 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.

#### 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. It 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.

#### 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 **TIMEZONE** offsets stored in the data.

#### TIMESTAMP WITH LOCAL TIME ZONE

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

**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.

**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 default is 6.

Avoiding 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.

**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.

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: Chapter 12 for information about using these Oracle-defined C data types

Data Conversions

Table 3–5 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 REFs on input.
- Named data types stored in the database can be converted to SQLT_NTY (and represented by a C struct in the application) on output.
- SQLT_NTY (represented by a C struct in an application) is converted to the internal representation of the corresponding type on input.

LOBs are shown in Table 3–6, because of the width limitation.

See Also: Chapter 12 for information about OCIString, OCINumber, and other new data types

Table 3–5 Data Conversions

<table>
<thead>
<tr>
<th>NA1</th>
<th>INTERNAL DATA TYPES -&gt;</th>
<th>NA</th>
<th>NA</th>
<th>NA</th>
<th>NA</th>
<th>NA</th>
<th>NA</th>
<th>NA</th>
<th>NA</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTERNAL DATA TYPES</td>
<td>VARCHAR2</td>
<td>NUMBER</td>
<td>LONG</td>
<td>ROWID</td>
<td>UROWID</td>
<td>DATE</td>
<td>RAW</td>
<td>LONG</td>
<td>RAW</td>
<td>CHAR</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>I/O²</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O³</td>
<td>I/O⁴</td>
<td>1/O⁵</td>
<td>1/O⁵</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMBER</td>
<td>I/O⁶</td>
<td>I/O</td>
<td>I⁷</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>1/O⁶</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>I/O⁶</td>
<td>I/O</td>
<td>I</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>1/O⁶</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>I/O⁶</td>
<td>I/O</td>
<td>I</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>1/O⁶</td>
<td></td>
</tr>
<tr>
<td>STRING</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O³</td>
<td>I/O³</td>
<td>I/O⁴</td>
<td>1/O⁵</td>
<td>1/O²⁸</td>
<td>I/O</td>
<td></td>
</tr>
<tr>
<td>VARNUM</td>
<td>I/O⁶</td>
<td>I/O</td>
<td>I</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>1/O⁶</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>I/O⁶</td>
<td>I/O</td>
<td>I</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>1/O⁶</td>
<td></td>
</tr>
<tr>
<td>LONG</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O³</td>
<td>I/O³</td>
<td>I/O⁴</td>
<td>1/O⁵</td>
<td>1/O²⁸</td>
<td>I/O</td>
<td></td>
</tr>
<tr>
<td>VARCHAR</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O³</td>
<td>I/O³</td>
<td>I/O⁴</td>
<td>1/O⁵</td>
<td>1/O²⁸</td>
<td>I/O</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>I/O</td>
<td>NA</td>
<td>I</td>
<td>NA</td>
<td>NA</td>
<td>I/O</td>
<td>NA</td>
<td>NA</td>
<td>I/O</td>
<td></td>
</tr>
</tbody>
</table>
Data Conversions

Table 3–6 shows the data conversions for LOBs. 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, VARBAR, LONG RAW, and LONG VARRAW) convert to the internal BLOB data type.

### Data Conversions for LOB Data Type Descriptors

<table>
<thead>
<tr>
<th>EXTERNAL DATA TYPES</th>
<th>INTERNAL CLOB</th>
<th>INTERNAL BLOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR</td>
<td>I/O&lt;sup&gt;1&lt;/sup&gt;</td>
<td>NA&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>CHAR</td>
<td>I/O</td>
<td>NA</td>
</tr>
<tr>
<td>LONG</td>
<td>I/O</td>
<td>NA</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>I/O</td>
<td>NA</td>
</tr>
<tr>
<td>RAW</td>
<td>NA</td>
<td>I/O</td>
</tr>
<tr>
<td>VARBAR</td>
<td>NA</td>
<td>I/O</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>NA</td>
<td>I/O</td>
</tr>
<tr>
<td>LONG VARRAW</td>
<td>NA</td>
<td>I/O</td>
</tr>
</tbody>
</table>

<sup>1</sup> I/O = Conversion is valid for input or output.
<sup>2</sup> NA = Conversion is not valid for input or output.
Data Conversions 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 3–7).

### Table 3–7 Data Conversion for Datetime and Interval Types

<table>
<thead>
<tr>
<th>External Types/Internal Types</th>
<th>VARCHAR, CHAR</th>
<th>DATE</th>
<th>TS</th>
<th>TSTZ</th>
<th>TSLTZ</th>
<th>INTERVAL YEAR TO MONTH</th>
<th>INTERVAL DAY TO SECOND</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2, CHAR</td>
<td>1/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
</tr>
<tr>
<td>DATE</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>NA2</td>
<td>NA</td>
</tr>
<tr>
<td>OCI DATE</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ANSI DATE</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>TIMESTAMP (TS)</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE (TSTZ)</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>I/O</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>I/O</td>
<td>NA</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>I/O</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>I/O</td>
</tr>
</tbody>
</table>

1 I/O = Conversion is valid for input or output.
2 NA means not applicable.

### 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.
Data Conversion Notes 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.

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.

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.

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.

Data Conversion for BINARY_FLOAT and BINARY_DOUBLE in OCI

Table 3–8 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>
<thead>
<tr>
<th>External Types/Internal Types</th>
<th>BINARY_FLOAT</th>
<th>BINARY_DOUBLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR</td>
<td>I/O¹</td>
<td>I/O</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>I/O</td>
<td>I/O</td>
</tr>
<tr>
<td>NUMBER</td>
<td>I/O</td>
<td>I/O</td>
</tr>
<tr>
<td>INTEGER</td>
<td>I/O</td>
<td>I/O</td>
</tr>
<tr>
<td>FLOAT</td>
<td>I/O</td>
<td>I/O</td>
</tr>
<tr>
<td>STRING</td>
<td>I/O</td>
<td>I/O</td>
</tr>
</tbody>
</table>
Table 3–9 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 3–9  Data Conversions for Internal to External Numeric Data Types

<table>
<thead>
<tr>
<th>Internal Types/External Types</th>
<th>BINARY_FLOAT</th>
<th>BINARY_DOUBLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR</td>
<td>I/O</td>
<td>I/O</td>
</tr>
<tr>
<td>NUMBER</td>
<td>I/O</td>
<td>I/O</td>
</tr>
<tr>
<td>LONG</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>CHAR</td>
<td>I/O</td>
<td>I/O</td>
</tr>
<tr>
<td>BINARY_FLOAT</td>
<td>I/O</td>
<td>I/O</td>
</tr>
<tr>
<td>BINARY_DOUBLE</td>
<td>I/O</td>
<td>I/O</td>
</tr>
</tbody>
</table>

1  An (I/O) implies that the conversion is valid for input and output (binds and defines)

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:

```sql
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 3–10 lists the possible values for an OCITypeCode. There is a value corresponding to each Oracle data type.

<table>
<thead>
<tr>
<th>Value</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_TYPECODE_REF</td>
<td>REF</td>
</tr>
<tr>
<td>OCI_TYPECODE_DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>OCI_TYPECODE_TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>OCI_TYPECODE_TIMESTAMP_TZ</td>
<td>TIMESTAMP WITH TIME ZONE</td>
</tr>
<tr>
<td>OCI_TYPECODE_TIMESTAMP_LTZ</td>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
</tr>
<tr>
<td>OCI_TYPECODE_INTERVAL_YM</td>
<td>INTERVAL YEAR TO MONTH</td>
</tr>
<tr>
<td>OCI_TYPECODE_INTERVAL_DS</td>
<td>INTERVAL DAY TO SECOND</td>
</tr>
<tr>
<td>OCI_TYPECODE_REAL</td>
<td>Single-precision real</td>
</tr>
<tr>
<td>OCI_TYPECODE_DOUBLE</td>
<td>Double-precision real</td>
</tr>
<tr>
<td>OCI_TYPECODE_FLOAT</td>
<td>Floating-point</td>
</tr>
<tr>
<td>OCI_TYPECODE_NUMBER</td>
<td>Oracle NUMBER</td>
</tr>
<tr>
<td>OCI_TYPECODE_BFLOAT</td>
<td>BINARY_FLOAT</td>
</tr>
<tr>
<td>OCI_TYPECODE_BDOUBLE</td>
<td>BINARY_DOUBLE</td>
</tr>
<tr>
<td>OCI_TYPECODE_DECIMAL</td>
<td>Decimal</td>
</tr>
<tr>
<td>OCI_TYPECODE_OCTET</td>
<td>Octet</td>
</tr>
<tr>
<td>OCI_TYPECODE_INTEGER</td>
<td>Integer</td>
</tr>
<tr>
<td>OCI_TYPECODE_SMALLINT</td>
<td>Small int</td>
</tr>
<tr>
<td>OCI_TYPECODE_RAW</td>
<td>RAW</td>
</tr>
<tr>
<td>OCI_TYPECODE_VARCHAR2</td>
<td>Variable string ANSI SQL, that is, VARCHAR2</td>
</tr>
<tr>
<td>OCI_TYPECODE_VARCHAR</td>
<td>Variable string Oracle SQL, that is, VARCHAR</td>
</tr>
<tr>
<td>OCI_TYPECODE_CHAR</td>
<td>Fixed-length string inside SQL, that is SQL CHAR</td>
</tr>
<tr>
<td>OCI_TYPECODE_VARRAY</td>
<td>Variable-length array (varray)</td>
</tr>
<tr>
<td>OCI_TYPECODE_TABLE</td>
<td>Multiset</td>
</tr>
<tr>
<td>OCI_TYPECODE_CLOB</td>
<td>Character large object (CLOB)</td>
</tr>
<tr>
<td>OCI_TYPECODE_BLOB</td>
<td>Binary large object (BLOB)</td>
</tr>
<tr>
<td>OCI_TYPECODE_BFILE</td>
<td>Binary large object file (BFILE)</td>
</tr>
<tr>
<td>OCI_TYPECODE_OBJECT</td>
<td>Named object type, or SYS.XMLType</td>
</tr>
<tr>
<td>OCI_TYPECODE_NAMEDCOLLECTION</td>
<td>Domain (named primitive type)</td>
</tr>
</tbody>
</table>
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` types 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 3–11 illustrates the mappings between `SQLT` and `OCI_TYPECODE` types.

<table>
<thead>
<tr>
<th>Oracle Type System Typename</th>
<th>Oracle Type System Type</th>
<th>Equivalent SQLT Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFILE</td>
<td>OCI_TYPECODE_BFILE</td>
<td>SQLT_BFILE</td>
</tr>
<tr>
<td>BLOB</td>
<td>OCI_TYPECODE_BLOB</td>
<td>SQLT_BLOB</td>
</tr>
<tr>
<td>CHAR (n)</td>
<td>OCI_TYPECODE_CHAR (n)</td>
<td>SQLT_AFC (n)</td>
</tr>
<tr>
<td>CLOB</td>
<td>OCI_TYPECODE_CLOB</td>
<td>SQLT_CLOB</td>
</tr>
<tr>
<td>COLLECTION</td>
<td>OCI_TYPECODE_NAMEDCOLLECTION</td>
<td>SQLT_NCO</td>
</tr>
<tr>
<td>DATE</td>
<td>OCI_TYPECODE_DATE</td>
<td>SQLT_DAT</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>OCI_TYPECODE_TIMESTAMP</td>
<td>SQLT_TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>OCI_TYPECODE_TIMESTAMP_TZ</td>
<td>SQLT_TIMESTAMP_TZ</td>
</tr>
<tr>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>OCI_TYPECODE_TIMESTAMP_LTZ</td>
<td>SQLT_TIMESTAMP_LTZ</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>OCI_TYPECODE_INTERVAL_YM</td>
<td>SQLT_INTERVAL_YM</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>OCI_TYPECODE_INTERVAL_DS</td>
<td>SQLT_INTERVAL_DS</td>
</tr>
<tr>
<td>FLOAT (b)</td>
<td>OCI_TYPECODE_FLOAT (b)</td>
<td>SQLT_FLT (8)</td>
</tr>
<tr>
<td>DECIMAL (p)</td>
<td>OCI_TYPECODE_DECIMAL (p)</td>
<td>SQLT_NUM (p, 0)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>OCI_TYPECODE_DOUBLE</td>
<td>SQLT_FLT (8)</td>
</tr>
<tr>
<td>BINARY_FLOAT</td>
<td>OCI_TYPECODE_BFLOAT</td>
<td>SQLT_BFLOAT</td>
</tr>
<tr>
<td>BINARY_DOUBLE</td>
<td>OCI_TYPECODE_BDOUBLE</td>
<td>SQLT_BDOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>OCI_TYPECODE_INTEGER</td>
<td>SQLT_INT (i)</td>
</tr>
<tr>
<td>NUMBER (p, s)</td>
<td>OCI_TYPECODE_NUMBER</td>
<td>SQLT_NUM (p, s)</td>
</tr>
<tr>
<td>OCTET</td>
<td>OCI_TYPECODE_OCTET</td>
<td>SQLT_INT (1)</td>
</tr>
<tr>
<td>POINTER</td>
<td>OCI_TYPECODE_PTR</td>
<td>&lt;NONE&gt;</td>
</tr>
<tr>
<td>RAW</td>
<td>OCI_TYPECODE_RAW</td>
<td>SQLT_LVB</td>
</tr>
<tr>
<td>REAL</td>
<td>OCI_TYPECODE_REAL</td>
<td>SQLT_FLT (4)</td>
</tr>
<tr>
<td>REF</td>
<td>OCI_TYPECODE_REF</td>
<td>SQLT_REF</td>
</tr>
<tr>
<td>OBJECT or SYS.XMLType</td>
<td>OCI_TYPECODE_OBJECT</td>
<td>SQLT_NTY</td>
</tr>
</tbody>
</table>
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.
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
- Preparing Statements
- Binding Placeholders in OCI
- Executing Statements
- Describing Select-List Items
- Defining Output Variables in OCI
- Fetching Results
- Using Scrollable Cursors in OCI

Overview of SQL Statement Processing

Chapter 2 discussed the basic steps involved in any OCI application. This chapter presents a more detailed look at the specific tasks involved in processing SQL statements in an OCI program.

One of the most common tasks of an OCI program is to accept and process SQL statements. This section outlines the specific steps involved in this processing.

Once you have allocated the necessary handles and connected to an Oracle database, follow the steps illustrated in Figure 4–1.
1. Prepare the statement. Define an application request using `OCIStmtPrepare2()` or `OCIStmtPrepare()`. `OCIStmtPrepare2()` is an enhanced version of `OCIStmtPrepare()` that was introduced to support statement caching.

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.
   - `OCIBindByPos()`
   - `OCIBindByName()`
   - `OCIBindObject()`
   - `OCIBindArrayOfStruct()`
   - `OCIBindDynamic()`

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 `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.
Preparing 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" on page 9-26

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() or OCIStmtPrepare() 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 4–1.

### Table 4–1 OCI_ATTR_STMT_TYPE Values and Statement Types

<table>
<thead>
<tr>
<th>Attribute Value</th>
<th>Statement Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_STMT_SELECT</td>
<td>SELECT statement</td>
</tr>
<tr>
<td>OCI_STMT_UPDATE</td>
<td>UPDATE statement</td>
</tr>
<tr>
<td>OCI_STMT_DELETE</td>
<td>DELETE statement</td>
</tr>
<tr>
<td>OCI_STMT_INSERT</td>
<td>INSERT statement</td>
</tr>
<tr>
<td>OCI_STMT_CREATE</td>
<td>CREATE statement</td>
</tr>
<tr>
<td>OCI_STMT_DROP</td>
<td>DROP statement</td>
</tr>
<tr>
<td>OCI_STMT.Alter</td>
<td>ALTER statement</td>
</tr>
</tbody>
</table>
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.

---

See Also:
- "Using PL/SQL in an OCI Program" on page 2-29
- "OCIStmtPrepare2()" on page 17-12 or "OCIStmtPrepare()" on page 17-10

---

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.

---

See Also:
- "Using PL/SQL in an OCI Program" on page 2-29
- "OCIStmtPrepare2()" on page 17-12 or "OCIStmtPrepare()" on page 17-10

---

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 4–1, 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 4–1  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)
```

**See Also:** Chapter 5 for detailed information about implementing bind operations

**Rules for 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.

**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.

**See Also:** "Defining Output Variables in OCI" on page 4-12

### 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.

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

It is not necessary to specify a `snapshot` when calling `OCIStmtExecute()`. The following sample code shows a basic execution in which the snapshot parameters are passed as NULL.

```
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0, (OCISnapshot *)NULL, (OCISnapshot *) NULL, OCI_DEFAULT));
```

**Note:** The `checkerr()` function, which is user-developed, evaluates the return code from an OCI application.

### Execution Modes of `OCIStmtExecute()`

You can specify a number of modes for the `OCIStmtExecute()` call. This section describes the `OCIStmtExecute()` call. See "`OCIStmtExecute()`" on page 17-3 for other values of the parameter `mode`.
Batch Error Mode

OCI provides the ability to perform array DML operations. For example, an application can process an array of `INSERT`, `UPDATE`, or `DELETE` statements with a single statement execution. If one of the operations fails due to an error from the server, such as a unique constraint violation, the array operation 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 Example 4–2.

**Example 4–2 Calling OCIAttrGet() to Retrieve the Number of Errors Encountered During an Array DML Operation**

```c
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 Example 4–3.

**Example 4–3 Retrieving Information About Each Error Following an Array DML Operation**

```c
OCIError errhdl1, errhp2;
for (i=0; i<num_errs; i++)
{
```
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.

**Example of Batch Error Mode**

Example 4–4 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 `stmtpl` and `stmtp2` to issue the INSERT and UPDATE statements, respectively.

```c
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 (stmtpl, &bindp1[0], errhp, 1, (void *)&number[0],
    sizeof(number[0]), SQLT_INT, (void *)0, (ub2 *)0, (ub2 *)0, 0,
    (ub4 *)0, OCI_DEFAULT);
OCIBindByPos (stmtpl, &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, stmtpl, 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 (stmtpl, 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_offset, 0,
            OCI_ATTR_DML_ROW_OFFSET, errhp2);
        /* get server diagnostics */
        OCIErrorGet (..., errhndl[i], ...);
    }
}
```
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:

```sql
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.
Describing Select-List Items

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 `OCIArrAttrGet()` 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 4–5 Implicit Describe - Select List Is Available as an Attribute of the Statement Handle 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()` or `OCIStmtPrepare()`.

Example 4–5  Implicit Describe - Select List Is Available as an Attribute of the Statement Handle

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

text *sqlstmt = (text *)'SELECT * FROM employees WHERE employee_id = 100'; 
checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt, 
{ub4)strlen((char *)sqlstmt), 
{ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT}); 
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, 0, 0, (OCISnapshot *)0, 
(OCISnapshot *)0, OCI_DEFAULT)); 
/* Request a parameter descriptor for position 1 in the select list */ 
counter = 1; 
parm_status = OCIParamGet((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, OCIArrAttrGet((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; 
... 
```

See Also: Chapter 6 for information about using `OCIDescribeAny()` to obtain metadata pertaining to schema objects
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 */
...

The checkerr() function in Example 4–5 is used for error handling. The complete listing can be found in the first sample application in Appendix B.

The calls to OCIAttrGet() and OCIParamGet() are local calls that do not require a network round-trip, because all of the select-list information is cached on the client side after the statement is executed.

See Also:
- "OCIParamGet()" on page 16-59
- "OCIArrayDescriptorAlloc()" on page 16-48
- "Parameter Attributes" on page 6-4 for a list of the specific attributes of the parameter descriptor that may be read by OCIArrayDescriptorAlloc() 

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 4–6 demonstrates the use of explicit describe in a select list to return information about columns.
Defining Output Variables in OCI

Example 4-6  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, OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
    (ub4)strlen((char *)sqlstmt),
    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

/* initialize svchp, stmhp, errhp, rowoff, iters, snap_in, snap_out */
/* set the execution mode to OCI_DESCRIBE_ONLY. Note that setting the mode to
OCI_DEFAULT does an implicit describe of the statement in addition to executing
the statement */
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, 0, 0,
    (OCISnapshot *) 0, (OCISnapshot *) 0, OCI_DESCRIBE_ONLY));

/* Get the number of columns in the query */
checkerr(errhp, OCIAttrGet((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 *)&numcols, OCI_ATTR_DATA_TYPE, errhp));
}
...

Defining Output Variables in OCI

Query statements return data from the database to your application. When processing
a query, you must define an output variable or an array of output variables for each
item in the select list 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:  Chapter 5, "Binding and Defining in OCI"
Fetching Results

If an OCI application has processed a query, it is typically necessary to fetch the results with `OCIStmtFetch2()` after the statement has completed execution. The `OCIStmtFetch2()` function supports scrollable cursors.

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

Fetched data is retrieved into output variables that have been specified by define operations.

**Note:** If output variables are defined for a `SELECT` statement before a call to `OCIStmtExecute()`, the number of rows specified by the `iters` parameter is fetched directly into the defined output buffers.

**See Also:**
- These statements mentioned previously fetch data associated with the sample code in "Steps Used in OCI Defining" on page 5-14. See that example for more information.
- "Overview of Defining in OCI" on page 5-13 for information about defining output variables

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:**
- Chapter 7 for more information about working with LOB locators in OCI
- "Defining LOB Output Variables" on page 5-16 for usage and examples of selecting LOB data without the use of locators

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`.

See Also: "Using Scrollable Cursors in OCI" on page 4-14
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` and nested cursor columns. By default, prefetching is not turned on for `REF CURSOR`. To turn on prefetching for `REF CURSOR`, 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`.

**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" on page A-27

---

**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.

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.

---

**See Also:**

- "OCIStmtFetch2()" on page 17-6
- "Setting Prefetch Count" on page 4-13

### Increasing Scrollable Cursor Performance

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

### Example of Access on a Scrollable Cursor

Assume that a result set is returned by the following SQL query, and that the table `EMP` has 14 rows:

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

One use of scrollable cursors is shown in Example 4–7.
Example 4-7 Access on a Scrollable Cursor

```c
/* 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));
}
...
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
- Binding and Defining Arrays of Structures in OCI
- 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
- Runtime Data Allocation and Piecewise Operations in OCI

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 5–1.

**See Also:** "Steps Used in OCI Binding" on page 5-5 for the code that implements this example
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, REFS 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)).

### Named Binds and Positional Binds

The SQL statement in Figure 5–1 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()` 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()` call.

To use the previous example for a positional bind:

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

The five placeholders are then each bound by calling `OCIBindByPos()` and passing the position number of the placeholder in the `position` parameter. For example, the `:empno` placeholder would be bound by calling `OCIBindByPos()` with a position of 1, `:ename` with a position of 2, and so on.
In 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:

```sql
SELECT empno FROM emp
WHERE sal > :some_value
AND comm > :some_value
```

An OCI application could complete the binds for this statement with a single call to `OCIBindByName()` to bind the `:some_value` placeholder by name. In this case, all bind placeholders for `:some_value` get assigned the same value as provided by the `OCIBindByName()` 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:

```sql
INSERT INTO emp VALUES
```

If you are using the `OCIBindByName()` 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()` call if your program evolves to add more bind variables in your statement text.

If you are using the `OCIBindByPos()` 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()` 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()` 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.

**OCI Array Interface**

You can pass data to the Oracle database in various ways.

You can execute a SQL statement repeatedly using the `OCISQExecute()` 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 `OCISQExecute()`. 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 `OCISqlStmtExecute()` 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 `OCISqlStmtExecute()` using an input array, which involves only one network round-trip.

**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).

**Binding Placeholders in PL/SQL**

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

When you bind placeholders in a PL/SQL block to program variables, you must use `OCIBindByName()` or `OCIBindByPos()` to perform the basic binds for host variables that are either scalars or arrays.

The following short PL/SQL block contains two placeholders, which represent IN parameters to a procedure that updates an employee's salary, when given the employee number and the new salary amount:

```sql
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:

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

In this block, you would use `OCIBindByName()` to bind variables in place of the :emp_name, :salary, and :commission output placeholders, and in place of the input placeholder :emp_number.

**Note:** All buffers, even pure OUT buffers, must be initialized by setting the buffer length to zero in the bind call, or by setting the corresponding indicator to -1.

**See Also:** "Information for Named Data Type and REF Binds" on page 12-26 for more information about binding PL/SQL placeholders.
Steps Used in OCI Binding

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

Once the bind is complete, the OCI library 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 5–1 shows handle allocation and binding for each placeholder in a SQL statement.

Example 5–1 Handle Allocation and Binding for Each Placeholder in a SQL Statement

```
/* The SQL statement, associated with stmthp (the statement handle) by calling OCIStmtPrepare() */
text *insert = (text *) "INSERT INTO emp(empno, ename, job, sal, deptno) VALUES (:empno, :ename, :job, :sal, :deptno)";

/* Bind the placeholders in the SQL statement, one per bind handle. */
checkerr(errhp, OCIBindByName(stmthp, &bnd1p, errhp, (text *) ":ENAME", strlen(":ENAME"), (ub1 *) ename, enamelen+1, SQLT_STR, (void *) 0, (ub2 *) 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, (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, (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, (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, (ub4 *) 0, OCI_DEFAULT));
```

Note: The checkerr() function evaluates the return code from an OCI application. The code for the function is in the Example for "OCIErrorGet()" on page 17-165.

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. This is explained in Defining PL/SQL Output Variables and in Information for Named Data Type and REF Defines, and PL/SQL OUT Binds.

Example 5–2 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 5–2  Defining a PL/SQL Statement to Be Used in OCI**

```c
/* 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;
    void     *tmp;
    OCISession *usrhp = (OCISession *)NULL;
    ... /* attach to Oracle database, and perform necessary initializations and authorizations */
    ...
    /* allocate a statement handle */
    checkerr(errhp, OCIHandleAlloc( (void *) envhp, (void **) &stmthp,
        OCI_HTYPE_STMT, 100, (void **) &tmp));
    /* prepare the statement request, passing the PL/SQL text block as the statement to be prepared */
    checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) give_raise, (ub4) strlen(give_raise), OCI_NTV_SYNTAX, OCI_DEFAULT));
    /* bind each of the placeholders to a program variable */
    checkerr( errhp, OCIBindByName(stmthp, &bnd1p, errhp, (text *) ":emp_number",
        -1, (ub1 *) &empno,
        (sword) sizeof(empno), SQLT_INT, (void *) 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, (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,
        ... /* perform other necessary actions */
    }
}
```

5-6 Oracle Call Interface Programmer’s Guide
Advanced Bind Operations in OCI

"Binding Placeholders in OCI" on page 4-4 discussed how a basic bind operation is performed to create an association between a placeholder in a SQL statement and a program variable by using OCIBindByName() or OCIBindByPos(). This section covers more advanced bind operations, including multistep binds, and binds of named data types and REFs.

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 5-1.

Table 5-1 Information Summary for Bind Types

<table>
<thead>
<tr>
<th>Type of Bind</th>
<th>Bind Data Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>Any scalar data type</td>
<td>Bind a single scalar using OCIBindByName() or OCIBindByPos().</td>
</tr>
<tr>
<td>Array of scalars</td>
<td>Any scalar data type</td>
<td>Bind an array of scalars using OCIBindByName() or OCIBindByPos().</td>
</tr>
<tr>
<td>Named data type</td>
<td>SQLT_NTY</td>
<td>Two bind calls are required:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ OCIBindByName() or OCIBindByPos()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ OCIBindObject()</td>
</tr>
<tr>
<td>REF</td>
<td>SQLT_REF</td>
<td>Two bind calls are required:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ OCIBindByName() or OCIBindByPos()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ OCIBindObject()</td>
</tr>
<tr>
<td>LOB</td>
<td>SQLT_BLOB</td>
<td>Allocate the LOB locator using OCIDescriptorAlloc(), and then bind its address, OCILobLocator **, with OCIBindByName() or OCIBindByPos(), by using one of the LOB data types.</td>
</tr>
<tr>
<td>BFFILE</td>
<td>SQLT_CLOB</td>
<td></td>
</tr>
</tbody>
</table>

```c
/* 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);
```
There are two ways of binding LOBs:

- Bind the LOB locator, rather than the actual LOB values. In this case the LOB value is written or read by passing a LOB locator to the OCI LOB functions.
- Bind the LOB value directly, without using the LOB locator.

### Binding LOB Locators

Either a single locator or an array of locators can be bound in a single bind call. In each case, the application must pass the address of a LOB locator and not the locator itself. For example, suppose that an application has prepared this SQL statement where `one_lob` is a bind variable corresponding to a LOB column:

```sql
INSERT INTO some_table VALUES (:one_lob)
```

Then your application makes the following declaration:

```c
OCILobLocator * one_lob;
```

Then the calls in Example 5–3 would be used to bind the placeholder and execute the statement:

```c
/* 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 */
```

You can also insert an array using the same SQL `INSERT` statement. In this case, the application would include the code shown in Example 5–4.
**Example 5–4  Binding the Placeholder and Executing the Statement to Insert an Array of Locators**

```c
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,...);
OCIDescAlloc(...,10,...);               /* 10 is the iters parameter */
```

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 BLOBs, CLOBs, and NCLOBs. Use `OCI_DTYPE_FILE` when allocating BFILES.

**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:** Chapter 7 for more information about the OCI LOB functions

**Binding LOB Data**

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

The bind of more than 4 kilobytes of data to a LOB column uses space from the temporary tablespace. 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 ...;
```

**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.

See Also: "Using the OCI_ATTR_MAXDATA_SIZE Attribute" on page 5-28

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 Example 5-5 illustrates this:

Example 5-5  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'));

Examples of Binding LOB Data  The following SQL statements are used in Example 5-6 through Example 5-13:

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 5–6**  Allowed: Inserting into C1, C2, and L Columns Up to 8000, 8000, and 2000 Byte-Sized Bind Variable Data Values, Respectively

```c
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)";
    OCStmtPrepare(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);
    OCStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

**Example 5–7**  Allowed: Inserting into C1 and L Columns up to 2000 and 8000 Byte-Sized Bind Variable Data Values, Respectively

```c
void insert() {
    /* The following is allowed */
    ub1 buffer[8000];
    text *insert_sql = (text *) "INSERT INTO lob_long_tab (C1, L) \ VALUES (:1, :2)";
    OCStmtPrepare(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);
    OCStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

**Example 5–8**  Allowed: Updating C1, C2, and L Columns up to 8000, 8000, and 2000 Byte-Sized Bind Variable Data Values, Respectively

```c
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";
    OCStmtPrepare(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);
    OCStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT);
}
Example 5–9  Allowed: Updating C1, C2, and L Columns up to 2000, 2000, and 8000 Byte-Sized Bind Variable Data Values, Respectively

```c
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 5–10  Allowed: Piecewise, Callback, and Array Insert or Update Operations

```c
void insert() {
    /* Piecewise, callback and array insert/update operations similar to 
     * the allowed regular insert/update operations are also allowed */
}
```

Example 5–11  Not Allowed: Inserting More Than 4000 Bytes into Both LOB and LONG Columns Using the Same INSERT Statement

```c
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);
    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 5–12  Allowed: Inserting into the CT3 LOB Column up to 2000 Byte-Sized Bind Variable Data Values

```c
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))";
```
Example 5–13 Not Allowed: Binding Any Length Data to a LOB Column in an Insert As Select Operation

```c
void insert()
{
    /* The following is NOT allowed because you cannot do insert as a*/
    /* 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, (ub4) OCI_DEFAULT);
    OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL,
                    (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

Binding in OCI_DATA_AT_EXEC Mode

If the `mode` parameter in a call to OCIBindByName() or OCIBindByPos() is set to OCI_DATA_AT_EXEC, an additional call to OCIBindDynamic() is necessary if the application 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" on page 5-33 for more information about piecewise operations

Binding REF CURSOR Variables

REF CURSORS 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" on page 5-32

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:

\[
\text{SELECT name, ssn FROM employees} \\
\text{WHERE empno = :empnum}
\]

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() 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.

**See Also:**
- Chapter 3 for more information about data types and conversions
- "Describing Select-List Items" on page 4-9

### Steps Used in OCI Defining

A basic define is done with a position call, OCIDefineByPos(). This step creates an
association between a select-list item and an output variable. Additional define calls
may be necessary for certain 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 5-14** shows a scalar output variable being defined following an execute and
describe operation.

**Example 5-14  Defining a Scalar Output Variable Following an Execute and Describe
Operation**

\[
\text{SELECT department_name FROM departments WHERE department_id = :dept_input}
\]

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

    printf("Enter employee dept: ");

---
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(); 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() are sufficient.

Oracle Database also provides predefined C data types that map object type attributes.
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.

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:

```sql
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:

```c
OCILobLocator * one_lob;
```

Then the following calls would be used to bind the placeholder and execute the statement:

```c
/* 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:

```c
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 BLOBs, CLOBs, and NCLOBs. Use `OCI_DTYPE_FILE` when allocating BFILES.

Defining LOB Data

Oracle Database allows nonzero defines for `SELECTs` 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 5–15 and Example 5–16:

```
CREATE TABLE lob_tab (C1 CLOB, C2 CLOB);
```

**Example 5–15  Defining LOBs Before Execution**

```c
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";

    OCStmtPrepare(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);
    OCStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *)0,
                  (OCISnapshot *)0, OCI_DEFAULT);
}
```

**Example 5–16  Defining LOBs After Execution**

```c
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";

    OCStmtPrepare(stmthp, errhp, select_sql, (ub4)strlen((char*)select_sql),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCStmtExecute(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);
    OCStmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT);
}
```

**Defining PL/SQL Output Variables**

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

**See Also:** "Information for Named Data Type and REF Defines, and PL/SQL OUT Binds" on page 12-27 for more information about defining PL/SQL output variables

**Defining for a Piecewise Fetch**

A piecewise fetch requires an initial call to OCIDefineByPos(). An additional call to OCIDefineDynamic() is necessary if the application uses callbacks rather than the standard polling mechanism.
Binding and Defining Arrays of Structures in OCI

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

Using arrays of structures can simplify the processing of 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.

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 5-2 shows how a skip parameter is determined. In this case, the skip parameter is the sum of the sizes of the fields field1 (2 bytes), field2 (4 bytes), and field3 (2 bytes), which is 8 bytes. This equals the size of one structure.

Figure 5-2 Determining Skip Parameters

Array of Structures

```
field 1   field 2   field 3   field 1   field 2   field 3   field 1   field 2   field 3  
2 bytes   4 bytes   2 bytes   2 bytes   4 bytes   2 bytes   2 bytes   4 bytes   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 \texttt{ub1} so that the \texttt{ub4} that begins the next structure in the array is properly aligned. In this case, the following statement may return an incorrect value:

\[
\text{skip\_parameter} = \text{sizeof} (\text{struct demo});
\]

On some operating systems this produces a proper skip parameter of 8. On other systems, \textit{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:

\[
\text{skip\_parameter} = \text{sizeof}(\text{demo\_array}[0]);
\]

**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:

\[
\text{text emp\_names}[4][20];
\]

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.

**OCI Calls Used with Arrays of Structures**

Two OCI calls must be used when you perform operations involving arrays of structures:

- Use \texttt{OCIBindArrayOfStruct()} for binding fields in arrays of structures for input variables
- Use \texttt{OCIDefineArrayOfStruct()} for defining arrays of structures for output variables.

\begin{footnotesize}
\textbf{Note:} Binding or defining for arrays of structures requires multiple calls. A call to \texttt{OCIBindByName()} or \texttt{OCIBindByPos()} must precede a call to \texttt{OCIBindArrayOfStruct()}, and a call to \texttt{OCIDefineByPos()} must precede a call to \texttt{OCIDefineArrayOfStruct()}.
\end{footnotesize}

**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 \texttt{OCIBindArrayOfStruct()} or \texttt{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.

\begin{footnotesize}
\textbf{See Also:} "Indicator Variables" on page 2-24
\end{footnotesize}
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:

```c
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() and OCIBindByName() 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:

```c
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() 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.

See Also:
- "OCIBindByName()" on page 16-64
- "OCIBindByPos()" on page 16-68
- "OCIDefineByPos()" on page 16-78

Example 5–17 illustrates the use of the structure OCIIOV and its mode values.

Example 5–17  Using Multiple Bind and Define Buffers

```c
/* 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);

...
DML with a RETURNING Clause in OCI

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

See Also:
- The Database demonstration programs included with your Oracle installation for complete examples. For additional information, see Appendix B.
- Oracle Database SQL Language Reference for more information about the use of the RETURNING clause with INSERT, UPDATE, or DELETE statements

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 col1 <= :high
RETURNING col1, col2, col3
INTO :out1, :out2, :out3
```

Binding RETURNING...INTO Variables

Because both the UPDATE and DELETE statements can affect multiple rows in the table, and a DML statement can be executed multiple times in a single 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 OCIBindByPos(), 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.

---

**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.

**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.
Binding the Output Variable

Binding the REF output variable in an OCI application requires three steps:

1. Set the initial bind information is set using OCIBindByName().
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().

The following pseudocode in Example 5–18 shows a function that performs the binds necessary for the preceding three steps.

Example 5–18  Binding the REF Output Variable in an OCI Application

```c
sword bind_output(stmthp, bndhp, errhp)
OCStmt *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_ERROR;
}

return OCI_SUCCESS;

Additional Notes About OCI Callbacks

When a callback function is called, the OCI_ATTR_ROWS_RETURNED attribute of the bind handle tells the application the number of rows being returned in that particular iteration. During the first callback of an iteration, you can allocate space for all rows that are returned for that bind variable. During subsequent callbacks of the same iteration, you increment the buffer pointer to the correct memory within the allocated space.

Array Interface for DML RETURNING Statements in OCI

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

If any of the 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.

Character Conversion in OCI Binding and Define

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

Choosing a Character Set

If a database column containing character data is defined to be an NCHAR or NVARCHAR2 column, then a bind or define involving that column must 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 a factor of four. Ensure that buffers that are provided to hold the data are of sufficient size.

In some cases, it may also be easier for an application to deal with NCHAR or NVARCHAR2 data in terms of numbers of characters, rather than numbers of bytes, which is the usual case.

Character Set Form and ID

Each OCI bind and define handle 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 Reference for more information about NCHAR data.

**Implicit Conversion Between CHAR and NCHAR**

As the result of implicit conversion between database character sets and national character sets, OCI can support cross binding and cross defining between CHAR and NCHAR. 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.

**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 5–19 illustrates the use of these functions (OCI provides a typedef called `utext` to facilitate binding and defining of UTF-16 data):

**Example 5–19 Setting the Client Character Set to OCI_UTF16ID in OCI**

```c
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;
OCIErrror *errhp;
OCIEnvNlsCreate(OCIEnv **envhp, ..., OCI_UTF16ID, ncsid);
...
OCIStmtPrepare(stmthp, ..., selstmt, ...); /* prepare UTF16 statement */
OCIDefineByPos(stmthp, defnp, ..., 1, thename, sizeof(thename), SQLT_CHR,...);
OCINlsEnvironmentVariableGet(&hdlcsid, (size_t)0, OCI_NLS_CHARSET_ID, (ub2)0, (size_t*)NULL);
OCIAttrSet(defnp, ..., &hdlcsid, 0, OCI_ATTR_CHARSET_ID, errhp);
/* change charset ID to NLS_LANG setting*/
...

See Also:
- "OCIEnvNlsCreate()" on page 16-17
- "OCINlsEnvironmentVariableGet()" on page 22-6

**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 (see Using the `OCI_ATTR_MAXDATA_SIZE` Attribute for more information).
- The `OCI_ATTR_MAXCHAR_SIZE` attribute sets the maximum number of characters allowed in the buffer on the server side (see Using the `OCI_ATTR_MAXCHAR_SIZE` Attribute for more information).

**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.

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() call.

Note: Regardless of the value of the attribute OCI_ATTR_MAXCHAR_SIZE, the buffer lengths specified in a bind or define call are always in terms of bytes. The actual length values sent and received by you are also in bytes.

Buffer Expansion During OCI Binding

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

If neither of these two attributes is set, OCI expands the buffer using its best estimates.

IN Binds  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.

Dynamic SQL For dynamic SQL, you can use the explicit describe to get OCI_ATTR_DATA_SIZE and OCI_ATTR_CHAR_SIZE in parameter handles, as a guide for setting OCI_ATTR_MAXDATA_SIZE and OCI_ATTR_MAXCHAR_SIZE attributes in bind handles. It is a good practice to specify OCI_ATTR_MAXDATA_SIZE and OCI_ATTR_MAXCHAR_SIZE to be no more than the actual column width in bytes or characters.

Buffer Expansion During Inserts Use OCI_ATTR_MAXDATA_SIZE to avoid unexpected behavior caused by buffer expansion during inserts.
Character Conversion in OCI Binding and Defining

Consider what happens when the database column has character-length semantics, and the user tries to insert data using `OCIBindByPos()` or `OCIBindByName()` while setting only the `OCI_ATTR_MAXCHAR_SIZE` to 3000 bytes. The database character set is UTF8 and the client character set is ASCII. Then, in this case although 3000 characters 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.

**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()` call can be used as the byte constraint.

**Dynamic SQL Selects**  When sizing buffers for dynamic SQL, always use the `OCI_ATTR_DATA_SIZE` value in the implicit describe to avoid data loss through truncation. If the database column is created using character-length semantics known through 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.

**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.

**General Compatibility Issues for Character-Length Semantics in OCI**

- 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.
Character Conversion in OCI Binding and Defining

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, as shown in Example 5–20. 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 5–20, 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 $M \times 30$ or $3 \times 30 = 90$, and $M \times 80$ or $3 \times 80 = 240$, respectively.

Example 5–20  Insert and Select Operations Using the OCI_ATTR_MAXCHAR_SIZE Attribute

```c
... 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 */ OCStmtPrepare(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)OCI_ATTR_MAXDATA_SIZE, errhp); /* or */ OCIAttrSet((void *)bnd1p, (ub4)OCI_HTYPE_BIND, (void *)&ename_max_chars, (ub4)OCI_ATTR_MAXCHAR_SIZE, errhp);
... /* Retrieving Column Data */ OCStmtPrepare(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);
... 
```

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.

See Also:  "Bind Handle Attributes" on page A-34
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 5–21 shows pseudocode that illustrates a bind and define for UTF-16 data.

**Example 5–21  Binding and Defining UTF-16 Data**

```c
...
OCStmt  *stmthp1, *stmthp2;
OCDefine *dfnlp1, *dfnlp2;
OCIBind *bndl1p, *bndl2p;
text *insstmt=
    (text *) "INSERT INTO EMP(ENAME, ADDRESS) VALUES (:ename, :address)";
text *selname =
    (text *) "SELECT ENAME, ADDRESS FROM EMP";
ute2 ename[21]; /* Name - UTF-16 */
ute2 address[51]; /* Address - UTF-16 */
ub4 ename_col_len = 20;
ub4 address_col_len = 50;
...
/* Inserting UTF-16 data */
OCStmtPrepare (stmthp1, errhp, insstmt, (ub4)strlen ((char *)insstmt),
    (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
OCIBindByName (stmthp1, &bndl1p, errhp, (text*)":ENAME",
    (sb4)strlen((char *)":ENAME"),
    (void *) ename, sizeof(ename), SQLT_STR,
    (void *)knsname_ind, (ub2 *) 0, (ub2 *) 0, (ub4) 0,
    (ub4 *)0, OCI_DEFAULT);
OCIAtrrSet ((void *) bndl1p, (ub4) OCI_HTYPE_BIND, (void *) &csid,
    (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAtrrSet ((void *) bndl1p, (ub4) OCI_HTYPE_BIND, (void *) &ename_col_len,
    (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving UTF-16 data */
OCStmtPrepare (stmthp2, errhp, selname, strlen((char *) selname),
    (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
OCDefineByPos (stmthp2, &dfnlp1, errhp, (ub4)1, (void *)ename,
    (sb4)sizeof(ename), SQLT_STR,
    (void *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);
OCIAtrrSet ((void *) dfnlp1, (ub4) OCI_HTYPE_DEFINE, (void *) &csid,
    (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
...
```

**PL/SQL REF CURSORs and Nested Tables in OCI**

OCI provides the ability to bind and define PL/SQL REF CURSORs and nested tables. An application can use a statement handle to bind and define these types of variables. As an example, consider this PL/SQL block:

```plsql
static const text *plsql_block = (text *)
    "begin \n        OPEN :cursor1 FOR SELECT employee_id, last_name, job_id, manager_id, \n            salary, department_id \n        ...
```
An application allocates a statement handle for binding by calling `OCIHandleAlloc()`, and then binds the \( \text{:cursor1} \) placeholder to the statement handle, as in the following code, where \( \text{:cursor1} \) is bound to \( \text{stm2p} \).

**Example 5–22  Binding the \( \text{:cursor1} \) Placeholder to the Statement Handle \( \text{stm2p} \) as a REF CURSOR**

```c
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)OCI_DEFAULT); 
```

In this code in **Example 5–22**, \( \text{stm1p} \) is the statement handle for the PL/SQL block, whereas \( \text{stm2p} \) 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:

```c
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 5–23**.

**Example 5–23  Defining a Nested Table (Second Position) as a Statement Handle**

```c
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); 
```

After execution, when you fetch a row into \( \text{stm2p} \) it becomes a valid statement handle.

---

**Note:** If you have retrieved multiple REF CURSORS, you must take care when fetching them into \( \text{stm2p} \). 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 \( \text{stm2p} \), you no longer have access to the data from the first REF CURSOR.

OCI does not support PL/SQL REF CURSORS that were executed in scrollable mode.

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

---

**Runtime Data Allocation and Piecewise Operations in OCI**

You can use 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 OCISstmtFetch2() 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 OCISstmtExecute() one time more than the number of pieces to be inserted (if callbacks are not used). This is because the first time OCISstmtExecute() 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 OCISstmtExecute() a total of n+1 times.

Similarly, when performing a piecewise fetch, you must call OCISstmtFetch2() once more than the number of pieces to be fetched.

**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.

**Types of Piecewise Operations**

You can perform piecewise operations in two ways:

- Use calls provided in the OCI library to execute piecewise operations under a polling paradigm.
- Employ user-defined callback functions to provide the necessary information and data blocks.

When you set the mode parameter of an OCIBindByPos() or OCIBindByName() call to OCI_DATA_AT_EXEC, it indicates that an OCI application is providing data for an INSERT or UPDATE operation dynamically at runtime.

Similarly, when you set the mode parameter of an OCIDefineByPos() 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.

### Providing INSERT or UPDATE Data at Runtime

When you specify the `OCI_DATA_AT_EXEC` mode in a call to `OCIBindByPos()` or `OCIBindByName()`, the `value_sz` parameter defines the total size of the data that can be provided at 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.

### 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 5–3 and expanded in the steps following the figure.

Figure 5–3  Performing Piecewise Insert

1. Initialize the OCI environment, allocate the necessary handles, connect to a server, authorize a user, and prepare a statement request by using OCIStmtPrepare().

2. Bind a placeholder by using OCIBindByName() or OCIBindByPos(). 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" on page 2-27
**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.

**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 `OCIBindByPos()`.

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.
See Also:
- "OCIBindByName()" on page 16-64
- "OCIBindByPos()" on page 16-68

Restrictions
The PL/SQL indexed table OCI binding interface does not support binding:
- Arrays of ADTs or REFS
- Arrays of descriptor types such as LOB descriptors, ROWID descriptors, datetime or interval descriptors
- Arrays of PLSQL record types

Providing FETCH Information at Run Time
When a call is made to OCIDefineByPos() with the mode parameter set to OCI_DYNAMIC_FETCH, an application can specify information about the data buffer at the time of fetch. You may also need to call OCIDefineDynamic() to set 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.

Performing a Piecewise Fetch
The fetch buffer can be of arbitrary size. In addition, each fetched piece does not need to be of the same size. The only requirement is that the size of the final fetch must be exactly the size of the last remaining piece. The size of each piece to be fetched is established by each OCIStmtSetPieceInfo() call. This process is illustrated in Figure 5-4 and explained in the steps following the figure.
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(), 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" on page 2-27

Piecewise Binds and Defines for LOBs

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

- 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()
    - OCIBindByName()
    - OCIBindByPos()
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()
  - OCIBindByName()
  - OCIBindByPos()

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

  **See Also:**
  - "OCILobRead2()" on page 17-73
  - "OCILobWrite2()" on page 17-81
  - "LOB Read and Write Callbacks" on page 7-11 for information about streaming using callbacks with OCILobWrite2() and OCILobRead2()
Describing Schema Metadata

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

This chapter contains these topics:

- Using OCIDescribeAny()
- Parameter Attributes
- Character-Length Semantics Support in Describe Operations
- Examples Using OCIDescribeAny()

Using OCIDescribeAny()

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

- Tables and views
- Synonyms
- Procedures
- Functions
- Packages
- Sequences
- Collections
- Types
- Schemas
- Databases

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

When an application describes a table, it can then retrieve information about that table's columns. Additionally, OCIDescribeAny() can directly describe subschema objects such as columns of a table, packages of a function, or fields of a type if given the name of the subschema object.

The OCIDescribeAny() call requires a describe handle as one of its arguments. The describe handle must be previously allocated with a call to OCIHandleAlloc().
The information returned by `OCIDescribeAny()` is organized hierarchically like a tree, as shown in Figure 6–1.

**Figure 6–1  OCIDescribeAny() Table Description**

The describe handle returned by 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()`.

---

**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 `OCISstmtExecute()`. Sometimes a column is not associated with a table. In most cases, the table is already known.
See Also:
- "Describing Select-List Items" on page 4-9
- "OCIDescribeAny()" on page 16-86

Notes on Types and Attributes

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

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 not 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" on page 3-6 for more information about SQLT_BOL
- "Typecodes" on page 3-25 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

Describing Types

To describe type objects, it is necessary to initialize the OCI process in object mode, as shown in Example 6–1.

Example 6–1  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()" on page 16-13

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.
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`:

```plaintext
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.

Parameter Attributes

A parameter is returned by `OCIParamGet()`. Parameters can describe different types of objects or information, and have attributes depending on the type of description they contain, or type-specific attributes. This section describes the attributes and handles that belong to different parameters.

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 6–1 lists the attributes of all parameters.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_OBJ_ID</td>
<td>Object or schema ID</td>
<td>ub4</td>
</tr>
<tr>
<td>OCI_ATTR_OBJ_NAME</td>
<td>Database name or object name in a schema</td>
<td>OraText *</td>
</tr>
</tbody>
</table>

Table 6–1 Attributes of All Parameters
The following sections list the attributes and handles specific to different types of parameters.

**Table or View Parameters**

*Table 6–2* lists the type-specific attributes for parameters for a table or view (type `OCI_PTYPE_TABLE` or `OCI_PTYPE_VIEW`).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_OBJID</td>
<td>Object ID</td>
<td>ub4</td>
</tr>
<tr>
<td>OCI_ATTR_NUM_COLS</td>
<td>Number of columns</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_LIST_COLUMNS</td>
<td>Column list (type <code>OCI_PTYPE_LIST</code>)</td>
<td>OCIParam *</td>
</tr>
<tr>
<td>OCI_ATTR_REF_TDO</td>
<td>REF to the type description object (TDO) of the base type for extent tables</td>
<td>OCIRef *</td>
</tr>
</tbody>
</table>
Table 6–3 lists additional attributes that belong to tables.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_IS_TEMPORARY</td>
<td>Indicates that the table is temporary</td>
<td>ubi</td>
</tr>
<tr>
<td>OCI_ATTR_IS_TYPED</td>
<td>Indicates that the table is typed</td>
<td>ubi</td>
</tr>
<tr>
<td>OCI_ATTR_DURATION</td>
<td>Duration of a temporary table. Values can be:</td>
<td>OCI_Duration</td>
</tr>
<tr>
<td></td>
<td>OCI_DURATION_SESSION - session</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCI_DURATION_TRANS - transaction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCI_DURATION_NULL - table not temporary</td>
<td></td>
</tr>
</tbody>
</table>

Table 6–4 lists the type-specific attributes when a parameter is for a procedure or function (type OCI_PTYPE_PROC or OCI_PTYPE_FUNC).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_LIST_ARGUMENTS</td>
<td>Argument list. See &quot;List Attributes&quot; on page 6-14.</td>
<td>void *</td>
</tr>
<tr>
<td>OCI_ATTR_IS_INVOKER_RIGHTS</td>
<td>Indicates that the procedure or function has invoker’s rights</td>
<td>ubi</td>
</tr>
</tbody>
</table>

Table 6–5 lists the attributes that are defined only for package subprograms.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_NAME</td>
<td>Name of the procedure or function</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_OVERLOAD_ID</td>
<td>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.</td>
<td>ub2</td>
</tr>
</tbody>
</table>

Package Attributes

Table 6–6 lists the attributes when a parameter is for a package (type OCI_PTYPE_PKG).
Table 6–6  Attributes of Packages

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_LIST_SUBPROGRAMS</td>
<td>Subprogram list. See “List Attributes” on page 6-14.</td>
<td>void *</td>
</tr>
<tr>
<td>OCI_ATTR_IS_INVOKER_RIGHTS</td>
<td>Indicates that the package has invoker’s rights?</td>
<td>ubl</td>
</tr>
</tbody>
</table>

Table 6–7  Attributes of Types

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_REF_TDO</td>
<td>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().</td>
<td>OCIRef *</td>
</tr>
<tr>
<td>OCI_ATTR_TYPECODE</td>
<td>Typecode. See “Data Type Codes” on page 6-3. Currently can be only OCI_TYPECODE_OBJECT or OCI_TYPECODE_NAMEDCOLLECTION.</td>
<td>OCITypeCode</td>
</tr>
<tr>
<td>OCI_ATTR_COLLECTION_TYPECODE</td>
<td>Typecode of collection if type is collection; invalid otherwise. See “Data Type Codes” on page 6-3. Currently can be only OCI_TYPECODE_VARRAY or OCI_TYPECODE_TABLE. If this attribute is queried for a type that is not a collection, an error is returned.</td>
<td>OCITypeCode</td>
</tr>
<tr>
<td>OCI_ATTR_IS_INCOMPLETE_TYPE</td>
<td>Indicates that this is an incomplete type</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_SYSTEM_TYPE</td>
<td>Indicates that this is a system type</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_PREDEFINED_TYPE</td>
<td>Indicates that this is a predefined type</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_TRANSIENT_TYPE</td>
<td>Indicates that this is a transient type</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_SYSTEM_GENERATED_TYPE</td>
<td>Indicates that this is a system-generated type</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_HAS_NESTED_TABLE</td>
<td>This type contains a nested table attribute.</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_HAS_LOB</td>
<td>This type contains a LOB attribute.</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_HAS_FILE</td>
<td>This type contains a BFILE attribute.</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_COLLECTION_ELEMENT</td>
<td>Handle to collection element. See “Collection Attributes” on page 6-10.</td>
<td>void *</td>
</tr>
<tr>
<td>OCI_ATTR_NUM_TYPE_ATTRS</td>
<td>Number of type attributes</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_LIST_TYPE_ATTRS</td>
<td>List of type attributes. See “List Attributes” on page 6-14.</td>
<td>void *</td>
</tr>
<tr>
<td>OCI_ATTR_NUM_TYPE_METHODS</td>
<td>Number of type methods</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_LIST_TYPE_METHODS</td>
<td>List of type methods. See “List Attributes” on page 6-14.</td>
<td>void *</td>
</tr>
<tr>
<td>OCI_ATTR_MAP_METHOD</td>
<td>Map method of type. See “Type Method Attributes” on page 6-9.</td>
<td>void *</td>
</tr>
</tbody>
</table>
Table 6–7 (Cont.) Attributes of Types

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_ORDER_METHOD</td>
<td>Order method of type. See &quot;Type Method Attributes&quot; on page 6-9.</td>
<td>void *</td>
</tr>
<tr>
<td>OCI_ATTR_IS_INVOKER_RIGHTS</td>
<td>Indicates that the type has invoker's rights</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_NAME</td>
<td>A pointer to a string that is the type attribute name</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_SCHEMA_NAME</td>
<td>A string with the schema name where the type has been created</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_IS_FINAL_TYPE</td>
<td>Indicates that this is a final type</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_INSTANTIABLE_TYPE</td>
<td>Indicates that this is an instantiable type</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_SUBTYPE</td>
<td>Indicates that this is a subtype</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_SUPERTYPE_SCHEMA_NAME</td>
<td>Name of the schema that contains the supertype</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_SUPERTYPE_NAME</td>
<td>Name of the supertype</td>
<td>OraText *</td>
</tr>
</tbody>
</table>

Table 6–8 Attributes of Type Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_DATA_SIZE</td>
<td>The maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERS.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR&gt;TypeCODE</td>
<td>Typecode. See &quot;Data Type Codes&quot; on page 6-3.</td>
<td>OCITypeCode</td>
</tr>
<tr>
<td>OCI_ATTR_DATA_TYPE</td>
<td>The data type of the type attribute. See &quot;Data Type Codes&quot; on page 6-3.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_NAME</td>
<td>A pointer to a string that is the type attribute name</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_PRECISION</td>
<td>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.</td>
<td>ub1 for explicit describe sb2 for implicit describe</td>
</tr>
<tr>
<td>OCI_ATTR_SCALE</td>
<td>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.</td>
<td>sb1</td>
</tr>
<tr>
<td>OCI_ATTR_TYPE_NAME</td>
<td>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.</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_SCHEMA_NAME</td>
<td>A string with the schema name under which the type has been created</td>
<td>OraText *</td>
</tr>
</tbody>
</table>
Table 6–8  (Cont.) Attributes of Type Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_REF_TDO</td>
<td>Returns the in-memory REF of the TDO for the type, if the column type is an</td>
<td>OCIRef *</td>
</tr>
<tr>
<td></td>
<td>object type. If space has not been reserved for the OCIRef, then it is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>allocated implicitly in the cache. The caller can then pin the TDO with</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCIObjectPin().</td>
<td></td>
</tr>
<tr>
<td>OCI_ATTR_CHARSET_ID</td>
<td>The character set ID, if the type attribute is of a string or character type</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_CHARSET_FORM</td>
<td>The character set form, if the type attribute is of a string or character type</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_FSPRECISION</td>
<td>The fractional seconds precision of a datetime or interval</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_LFPRECISION</td>
<td>The leading field precision of an interval</td>
<td>ub1</td>
</tr>
</tbody>
</table>

Table 6–9  Attributes of Type Methods

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_NAME</td>
<td>Name of method (procedure or function)</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_ENCAPSULATION</td>
<td>Encapsulation level of the method (either OCI_</td>
<td>OCITypeEncap</td>
</tr>
<tr>
<td></td>
<td>TYPEENCAP_PRIVATE or OCI_TYPEENCAP_PUBLIC)</td>
<td></td>
</tr>
<tr>
<td>OCI_ATTR_LIST_ARGUMENTS</td>
<td>Argument list. See &quot;OCI_ATTR_LIST_ARGUMENTS</td>
<td>void *</td>
</tr>
<tr>
<td></td>
<td>Attribute&quot; on page 6-4, and &quot;List Attributes&quot; on</td>
<td></td>
</tr>
<tr>
<td></td>
<td>page 6-14.</td>
<td></td>
</tr>
<tr>
<td>OCI_ATTR_IS_CONSTRUCTOR</td>
<td>Indicates that method is a constructor</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_DESTRUCTOR</td>
<td>Indicates that method is a destructor</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_OPERATOR</td>
<td>Indicates that method is an operator</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_SELFISH</td>
<td>Indicates that method is selfish</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_MAP</td>
<td>Indicates that method is a map method</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_ORDER</td>
<td>Indicates that method is an order method</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_RNDS</td>
<td>Indicates that &quot;Read No Data State&quot; is set for</td>
<td>ub1</td>
</tr>
<tr>
<td></td>
<td>method</td>
<td></td>
</tr>
<tr>
<td>OCI_ATTR_IS_RNPS</td>
<td>Indicates that &quot;Read No Process State&quot; is set</td>
<td>ub1</td>
</tr>
<tr>
<td></td>
<td>for method</td>
<td></td>
</tr>
<tr>
<td>OCI_ATTR_IS_WNDS</td>
<td>Indicates that &quot;Write No Data State&quot; is set for</td>
<td>ub1</td>
</tr>
<tr>
<td></td>
<td>method</td>
<td></td>
</tr>
<tr>
<td>OCI_ATTR_IS_WNPS</td>
<td>Indicates that &quot;Write No Process State&quot; is set</td>
<td>ub1</td>
</tr>
<tr>
<td></td>
<td>for method</td>
<td></td>
</tr>
<tr>
<td>OCI_ATTR_IS_FINAL_METHOD</td>
<td>Indicates that this is a final method</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_INSTANTIABLE_METHOD</td>
<td>Indicates that this is an instantiable method</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_OVERRIDING_METHOD</td>
<td>Indicates that this is an overriding method</td>
<td>ub1</td>
</tr>
</tbody>
</table>
Parameter Attributes

Collection Attributes

Table 6–10 lists the attributes when a parameter is for a collection type (type OCI_PTYPE_COLL).

### Table 6–10 Attributes of Collection Types

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_DATA_SIZE</td>
<td>The maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERS.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_TYPECODE</td>
<td>Typecode. See &quot;Data Type Codes&quot; on page 6-3.</td>
<td>OCITypeCode</td>
</tr>
<tr>
<td>OCI_ATTR_DATA_TYPE</td>
<td>The data type of the type attribute. See &quot;Data Type Codes&quot; on page 6-3.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_NUM_ELEMS</td>
<td>The number of elements in an array. It is only valid for collections that are arrays.</td>
<td>ub4</td>
</tr>
<tr>
<td>OCI_ATTR_NAME</td>
<td>A pointer to a string that is the type attribute name</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_PRECISION</td>
<td>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.</td>
<td>ub1 for explicit describe, sb2 for implicit describe</td>
</tr>
<tr>
<td>OCI_ATTR_SCALE</td>
<td>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.</td>
<td>sb1</td>
</tr>
<tr>
<td>OCI_ATTR_TYPE_NAME</td>
<td>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.</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_SCHEMA_NAME</td>
<td>A string with the schema name under which the type has been created</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_REF_TDO</td>
<td>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().</td>
<td>OCIRef *</td>
</tr>
<tr>
<td>OCI_ATTR_CHARSET_ID</td>
<td>The character set ID, if the type attribute is of a string or character type</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_CHARSET_FORM</td>
<td>The character set form, if the type attribute is of a string or character type</td>
<td>ub1</td>
</tr>
</tbody>
</table>

Synonym Attributes

Table 6–11 lists the attributes when a parameter is for a synonym (type OCI_PTYPE_SYN).
### Sequence Attributes

Table 6–12 lists the attributes when a parameter is for a sequence (type `OCI_PTYPE_SEQ`).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_OBJID</td>
<td>Object ID</td>
<td>ub4</td>
</tr>
<tr>
<td>OCI_ATTR_MIN</td>
<td>Minimum value (in Oracle NUMBER format)</td>
<td>ub1 *</td>
</tr>
<tr>
<td>OCI_ATTR_MAX</td>
<td>Maximum value (in Oracle NUMBER format)</td>
<td>ub1 *</td>
</tr>
<tr>
<td>OCI_ATTR_INCR</td>
<td>Increment (in Oracle NUMBER format)</td>
<td>ub1 *</td>
</tr>
<tr>
<td>OCI_ATTR_CACHE</td>
<td>Number of sequence numbers cached; zero if the sequence is not a cached sequence (in Oracle NUMBER format)</td>
<td>ub1 *</td>
</tr>
<tr>
<td>OCI_ATTR_ORDER</td>
<td>Whether the sequence is ordered</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_HW_MARK</td>
<td>High-water mark (in NUMBER format)</td>
<td>ub1 *</td>
</tr>
</tbody>
</table>

**See Also:** "OCINumber Examples" on page 12-10

### Column Attributes

**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.

Table 6–13 lists the attributes when a parameter is for a column of a table or view (type `OCI_PTYPE_COL`).
Table 6–13  Attributes of Columns of Tables or Views

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_CHAR_USED</td>
<td>Returns the type of length semantics of the column. Zero (0) means byte-length semantics and 1 means character-length semantics. See &quot;Character-Length Semantics Support in Describe Operations&quot; on page 6-17.</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_CHAR_SIZE</td>
<td>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 &quot;Character-Length Semantics Support in Describe Operations&quot; on page 6-17.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_DATA_SIZE</td>
<td>The maximum size of the column. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERS.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_DATA_TYPE</td>
<td>The data type of the column. See &quot;Data Type Codes&quot; on page 6-3.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_NAME</td>
<td>A pointer to a string that is the column name</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_PRECISION</td>
<td>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.</td>
<td>ub1 for explicit describe, sb2 for implicit describe</td>
</tr>
<tr>
<td>OCI_ATTR_SCALE</td>
<td>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.</td>
<td>sb1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_NULL</td>
<td>Returns 0 if null values are not permitted for the column. Does not return a correct value for a CUBE or ROLLUP operation.</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_TYPE_NAME</td>
<td>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.</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_SCHEMA_NAME</td>
<td>Returns a string with the schema name under which the type has been created.</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_REF_TDO</td>
<td>The REF of the TDO for the type, if the column type is an object type</td>
<td>OCIRef *</td>
</tr>
<tr>
<td>OCI_ATTR_CHARSET_ID</td>
<td>The character set ID, if the column is of a string or character type</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_CHARSET_FORM</td>
<td>The character set form, if the column is of a string or character type</td>
<td>ub1</td>
</tr>
</tbody>
</table>

Argument and Result Attributes

Table 6–14 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_RESULT).
### Table 6–14  Attributes of Arguments and Results

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_NAME</td>
<td>Returns a pointer to a string that is the argument name</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_POSITION</td>
<td>The position of the argument in the argument list. Always returns zero.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR&gt;TypeCode</td>
<td>Typecode. See &quot;Data Type Codes&quot; on page 6-3.</td>
<td>OCITypeCode</td>
</tr>
<tr>
<td>OCI_ATTR_DATA_TYPE</td>
<td>The data type of the argument. See &quot;Data Type Codes&quot; on page 6-3.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_DATA_SIZE</td>
<td>The size of the data type of the argument. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERs.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_PRECISION</td>
<td>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.</td>
<td>b1 for explicit describe, sb2 for implicit describe</td>
</tr>
<tr>
<td>OCI_ATTR_SCALE</td>
<td>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.</td>
<td>sb1</td>
</tr>
<tr>
<td>OCI_ATTR_LEVEL</td>
<td>The data type levels. This attribute always returns zero.</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_HAS_DEFAULT</td>
<td>Indicates whether an argument has a default</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_LIST_ARGUMENTS</td>
<td>The list of arguments at the next level (when the argument is of a record or table type)</td>
<td>void *</td>
</tr>
<tr>
<td>OCI_ATTR_IOMODE</td>
<td>Indicates the argument mode: 0 is IN (OCI_TYPEPARAM_IN), 1 is OUT (OCI_TYPEPARAM_OUT), 2 is IN/OUT (OCI_TYPEPARAM_INOUT)</td>
<td>OCITypeParamMode</td>
</tr>
<tr>
<td>OCI_ATTR_RADIX</td>
<td>Returns a radix (if number type)</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_IS_NULL</td>
<td>Returns 0 if null values are not permitted for the column</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_TYPE_NAME</td>
<td>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.</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_SCHEMA_NAME</td>
<td>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</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_SUB_NAME</td>
<td>For SQLT_NTY or SQLT_REF, returns a string with the type name, for package local types</td>
<td>OraText *</td>
</tr>
</tbody>
</table>
List Attributes

When a parameter is for a list of columns, arguments, or subprograms (type OCI_PTYPE_LIST), it has the type-specific attributes and handles (parameters) shown in Table 6–15.

The list has an OCI_ATTR_LTYPE attribute that designates the list type. Table 6–15 lists the possible values and their lower bounds when traversing the list.

<table>
<thead>
<tr>
<th>List Attribute</th>
<th>Description</th>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_LTYPE_COLUMN</td>
<td>Column list</td>
<td>1</td>
</tr>
<tr>
<td>OCI_LTYPE_ARG_PROC</td>
<td>Procedure argument list</td>
<td>1</td>
</tr>
<tr>
<td>OCI_LTYPE_ARG_FUNC</td>
<td>Function argument list</td>
<td>0</td>
</tr>
<tr>
<td>OCI_LTYPE_SUBPRG</td>
<td>Subprogram list</td>
<td>0</td>
</tr>
<tr>
<td>OCI_LTYPE_TYPE_ATTR</td>
<td>Type attribute list</td>
<td>1</td>
</tr>
<tr>
<td>OCI_LTYPE_TYPE_METHOD</td>
<td>Type method list</td>
<td>1</td>
</tr>
<tr>
<td>OCI_LTYPE_TYPE_ARG_PROC</td>
<td>Type method without result argument list</td>
<td>0</td>
</tr>
<tr>
<td>OCI_LTYPE_TYPE_ARG_FUNC</td>
<td>Type method without result argument list</td>
<td>1</td>
</tr>
<tr>
<td>OCI_LTYPE_SCH_OBJ</td>
<td>Object list within a schema</td>
<td>0</td>
</tr>
<tr>
<td>OCI_LTYPE_DB_SCH</td>
<td>Schema list within a database</td>
<td>0</td>
</tr>
</tbody>
</table>

The list has an OCI_ATTR_NUM_PARAMS attribute, which tells the number of elements in the list.

Each list has LowerBound... OCI_ATTR_NUM_PARAMS parameters. LowerBound is the value in the Lower Bound column of Table 6–15. For a function argument list, position 0 has a parameter for the return value (type OCI_PTYPE_ARG).

Schema Attributes

Table 6–16 lists the attributes when a parameter is for a schema type (type OCI_PTYPE_SCHEMA).
Table 6–16  Attributes Specific to Schemas

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_LIST_OBJECTS</td>
<td>List of objects in the schema</td>
<td>OCIParam *</td>
</tr>
</tbody>
</table>

Database Attributes

Table 6–17 lists the attributes when a parameter is for a database type (type OCI_PTYPE_DATABASE).

Table 6–17  Attributes Specific to Databases

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_VERSION</td>
<td>Database version</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_CHARSET_ID</td>
<td>Database character set ID from the server handle</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_NCHARSET_ID</td>
<td>Database national character set ID from the server handle</td>
<td>ub2</td>
</tr>
<tr>
<td>OCI_ATTR_LIST_SCHEMAS</td>
<td>List of schemas (type OCI_PTYPE_SCHEMA) in the database</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_MAX_PROC_LEN</td>
<td>Maximum length of a procedure name</td>
<td>ub4</td>
</tr>
<tr>
<td>OCI_ATTR_MAX_COLUMN_LEN</td>
<td>Maximum length of a column name</td>
<td>ub4</td>
</tr>
<tr>
<td>OCI_ATTR_CURSOR_COMMIT_BEHAVIOR</td>
<td>How a COMMIT operation affects cursors and prepared statements in the database.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Values are:</td>
<td>ub1</td>
</tr>
<tr>
<td></td>
<td>OCI_CURSOR_OPEN - Preserve cursor state as before the commit operation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCI_CURSOR_CLOSED - Cursors are closed on COMMIT, but the application can still reexecute the statement without preparing it again.</td>
<td></td>
</tr>
<tr>
<td>OCI_ATTR_MAX_CATALOG_NAMELEN</td>
<td>Maximum length of a catalog (database) name</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_CATALOG_LOCATION</td>
<td>Position of the catalog in a qualified table.</td>
<td>ub1</td>
</tr>
<tr>
<td></td>
<td>Values are OCI_CL_START and OCI_CL_END.</td>
<td></td>
</tr>
<tr>
<td>OCI_ATTR_SAVEPOINT_SUPPORT</td>
<td>Does database support savepoints? Values are OCI_SP_SUPPORTED and OCI_SP_UNSUPPORTED.</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_NOWAIT_SUPPORT</td>
<td>Does database support the nowait clause? Values are OCI_NW_SUPPORTED and OCI_NW_UNSUPPORTED.</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_AUTOCOMMIT_DDL</td>
<td>Is autocommit mode required for DDL statements? Values are OCI_AC_DDL and OCI_NO_AC_DDL.</td>
<td>ub1</td>
</tr>
<tr>
<td>OCI_ATTR_LOCKING_MODE</td>
<td>Locking mode for the database. Values are OCI_LOCK_IMMEDIATE and OCI_LOCK_DELAYED.</td>
<td>ub1</td>
</tr>
</tbody>
</table>

Rule Attributes

Table 6–18 lists the attributes when a parameter is for a rule (type OCI_PTYPE_RULE).
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_CONDITION</td>
<td>Rule condition</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_EVAL_CONTEXT_OWNER</td>
<td>Owner name of the evaluation context associated with the rule, if any</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_EVAL_CONTEXT_NAME</td>
<td>Object name of the evaluation context associated with the rule, if any</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_COMMENT</td>
<td>Comment associated with the rule, if any</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_LIST_ACTION_CONTEXT</td>
<td>List of name-value pairs in the action context (type OCI_PTYPE_LIST)</td>
<td>void *</td>
</tr>
</tbody>
</table>

**Rule Set Attributes**

Table 6–19 lists the attributes when a parameter is for a rule set (type OCI_PTYPE_RULE_SET).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_EVAL_CONTEXT_OWNER</td>
<td>Owner name of the evaluation context associated with the rule set, if any</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_EVAL_CONTEXT_NAME</td>
<td>Object name of the evaluation context associated with the rule set, if any</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_COMMENT</td>
<td>Comment associated with the rule set, if any</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_LIST_RULES</td>
<td>List of rules in the rule set (type OCI_PTYPE_LIST)</td>
<td>void *</td>
</tr>
</tbody>
</table>

**Evaluation Context Attributes**

Table 6–20 lists the attributes when a parameter is for an evaluation context (type OCI_PTYPE_EVALUATION_CONTEXT).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_EVALUATION_FUNCTION</td>
<td>Evaluation function associated with the evaluation context, if any</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_COMMENT</td>
<td>Comment associated with the evaluation context, if any</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_LIST_TABLE_ALIASES</td>
<td>List of table aliases in the evaluation context (type OCI_PTYPE_LIST)</td>
<td>void *</td>
</tr>
<tr>
<td>OCI_ATTR_LIST_VARIABLE_TYPES</td>
<td>List of variable types in the evaluation context (type OCI_PTYPE_LIST)</td>
<td>void *</td>
</tr>
</tbody>
</table>

**Table Alias Attributes**

Table 6–21 lists the attributes when a parameter is for a table alias (type OCI_PTYPE_TABLE_ALIAS).
Character-Length Semantics Support in Describe Operations

Table 6–21 Attributes Specific to Table Aliases

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_NAME</td>
<td>Table alias name</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_TABLE_NAME</td>
<td>Table name associated with the alias</td>
<td>OraText *</td>
</tr>
</tbody>
</table>

Variable Type Attributes

Table 6–22 lists the attributes when a parameter is for a variable (type OCI_PTYPE_VARIABLE_TYPE).

Table 6–22 Attributes Specific to Variable Types

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_NAME</td>
<td>Variable name</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_TYPE</td>
<td>Variable type</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_VAR_VALUE_FUNCTION</td>
<td>Variable value function associated with the variable, if any</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_VAR_METHOD_FUNCTION</td>
<td>Variable method function associated with the variable, if any</td>
<td>OraText *</td>
</tr>
</tbody>
</table>

Name Value Attributes

Table 6–23 lists the attributes when a parameter is for a name-value pair (type OCI_PTYPE_NAME_VALUE).

Table 6–23 Attributes Specific to Name-Value Pair

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_ATTR_NAME</td>
<td>Name</td>
<td>OraText *</td>
</tr>
<tr>
<td>OCI_ATTR_VALUE</td>
<td>Value</td>
<td>OCIAnyData*</td>
</tr>
</tbody>
</table>

Character-Length Semantics Support in Describe Operations

Since release Oracle9i, 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 OCISstmtExecute(). Other schema elements must be described explicitly through OCIDescribeAny().
Implicit Describing

If the database column was created using character-length semantics, then the implicit describe information 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`.

Explicit Describing

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.

See Also: "IN Binds" on page 5-29

Client and Server Compatibility Issues for Describing

When an Oracle9i or later client talks to an Oracle8i or earlier server, it behaves as if the database is only using byte-length semantics.

When an Oracle8i or earlier client talks to a Oracle9i 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 Oracle8i or earlier software release.

Examples Using OCIDescribeAny()

The following examples demonstrate the use of OCIDescribeAny() for describing different types of schema objects. For a more detailed code sample, see the demonstration program `cdemodsa.c` included with your Oracle Database installation.

See Also: Appendix B for additional information about the demonstration programs
Retrieving Column Data Types for a Table

**Example 6–2** illustrates the use of an explicit describe that retrieves the column data types for a table.

**Example 6–2  Using an Explicit Describe to Retrieve Column Data Types for a Table**

```c
...  
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 *)parmh, OCI_HTYPE_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;
    
... 
```

(Continued on next page)
Examples Using OCIDescribeAny()

Describing the Stored Procedure

The difference between a procedure and a function is that the latter has a return type at position 0 in the argument list, whereas the former has no argument associated with position 0 in the argument list. The steps required to describe type methods (also divided into functions and procedures) are identical to those of regular PL/SQL functions and procedures. 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 6–3, 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 level arguments.

Example 6–3 Describing the Stored Procedure

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 argl 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);
...
Retrieving Attributes of an Object Type

Example 6–4 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 6–4 Using an Explicit Describe on a Named Object Type

```c
... 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;
...```

"..."
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);
...

Retrieving the Collection Element's Data Type of a Named Collection Type

Example 6–5 illustrates the use of an explicit describe on a named collection type.

Example 6–5 Using an Explicit Describe on a Named Collection Type

text type_name[] = "phone_list_typ";
text type_name_len = (text) strlen((char*)type_name);
OCIRef *type_ref = OCIRef() 0;
```c
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_OTYPE_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_OTYPE_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_OTYPE_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_OTYPE_PARAM,
            (size_t)num_elements,
            OCI_ATTR_COLLECCION_ELEMENT_NUM, errh))
            return OCI_ERROR;
    }
```
Examples Using OCIDescribeAny()

Describing Schema Metadata

6-25

Examples Using OCIDescribeAny()

Describing with Character-Length Semantics

Example 6–6 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 OCIStmtPrepare().

Example 6–6 Using a Parameter Descriptor to Retrieve the Data Types, Column Names, and Character-Length Semantics

...\n
Describing Schema Metadata 6-25
/* 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 */
...
This chapter contains these topics:

- Using OCI Functions for LOBs
- Creating and Modifying Persistent LOBs
- Associating a BFILE in a Table with an Operating System File
- LOB Attributes of an Object
- Array Interface for LOBs
- 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

**Using OCI Functions for LOBs**

OCI includes a set of functions for performing operations on large objects (LOBs) in a database. Persistent LOBs (BLOBs, CLOBs, NCLOBs) are stored in the database tablespaces in a way that optimizes space and provides efficient access. These LOBs have the full transactional support of the Oracle database. BFILEs 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.

BFILEs are read-only. Oracle Database supports only binary BFILEs.

**See Also:**

- Appendix B 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
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 10g Release 2. The data interface only supports data size up to 2 GB – 1, the maximum size of an sb4 data type.

  See Also: Oracle Database SecureFiles and Large Objects Developer’s Guide chapter about data interface for persistent LOBs for more information and examples

- 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.

  See Also: "Binding LOB Data" on page 5-9 for usage and examples for both INSERT and UPDATE

For any LOB write command to be successful, a transaction must be open. If you commit a transaction before writing the data, you must lock the row again (by reissuing the SELECT...FOR UPDATE statement, for example), because the commit closes the transaction.

Associating a BFILE in a Table with an Operating System File

The BFILENAME function can be used in an INSERT statement to associate an external server-side (operating system) file with a BFILE column or attribute in a table. Using BFILENAME in an UPDATE statement associates the BFILE column or attribute with a different operating system file. OCILobFileSetName() can also be used to associate a BFILE in a table with an operating system file. BFILENAME is usually used in an INSERT or UPDATE statement without bind variables, and OCILobFileSetName() is used for bind variables.
LOB Attributes of an Object

An OCI application can use the OCIObjectNew() function to create a persistent or transient object with a LOB attribute.

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 OCILobWrite2() using the LOB locator in the object to write the data.

See Also: Chapter 11 and the chapters that follow it for more information about objects

There is a second way of writing to a LOB attribute. When using the data interface, you can bind or define character data for a CLOB attribute or RAW data for a BLOB attribute.

See Also:

- "Binding LOB Data" on page 5-9 for usage and examples for both INSERT and UPDATE statements
- "Defining LOB Data" on page 5-16 for usage and examples of SELECT statements

Transient Objects with LOB Attributes

An application can call OCIObjectNew() and create a transient object with an internal LOB (BLOB, CLOB, NLOB) 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.

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.

See Also:

- "OCILobFileSetName()" on page 17-52
- Oracle Database SecureFiles and Large Objects Developer's Guide for more information about the BFILENAME function
Using the data interface

You can bind or define arrays of character data for a CLOB column or RAW data for a BLOB column. You can use array bind and define interfaces to insert and select multiple rows with LOBs in one round-trip to the server.

See Also:

- "Binding LOB Data" on page 5-9 for usage and examples for both INSERT and UPDATE statements
- "Defining LOB Data" on page 5-16 for usage and examples of SELECT statements

Using the LOB locator

When using the LOB locator you must allocate the descriptors, as shown in Example 7–1.

Example 7–1 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], ...);
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 BLOBs, CLOBs, and NCLOBs. The number of bytes stored in a chunk is actually less than the size of the CHUNK parameter due to internal storage overhead. The function OCILobGetStorageLimit() is provided to return the maximum size in bytes of internal LOBs in the current installation.

Note: Oracle Database does not support BFILEs larger than 4 gigabytes in any programmatic environment. An additional file size limit imposed by your operating system also applies to BFILEs.

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 10g Release 1. Other changes were made in the read and write functions (OCILobRead2(), OCILobWrite2(), and OCILobWriteAppend2()) to solve several problems:

Problem: Before Oracle Database 10g 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_amtp parameter is ignored for both input and output.

Problem: For OCILobRead2(), there is no flag to indicate polling mode. There is no easy way for the users to say “I have a 100-byte buffer. Fill it as much as you can.” Previously, they had to estimate how many characters to specify for the amount. If they guessed too much, they were forced into polling mode unintentionally. The user code thus can get trapped in the polling mode and subsequent OCI calls are all blocked.

Solution: This call should take piece as an input parameter and if OCI_ONE_PIece is passed, it should fill the buffer as much as possible and come out even if the amount indicated by 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.
Using LOBs of Size Greater than 4 GB

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" on page 17-17
- "OCILobRead2()" on page 17-73
- "OCILobWrite2()" on page 17-81
- "OCILobWriteAppend2()" on page 17-85

Compatibility and Migration

Existing OCI programs can be enhanced to process larger amounts of LOB data that are greater than 4 GB. Table 7–1 summarizes compatibility issues in this table, "old" refers to releases before Oracle Database 10g Release 1, and NA means not applicable.

Table 7–1  LOB Functions Compatibility and Migration

<table>
<thead>
<tr>
<th>LOB Function</th>
<th>Old Client/New or Old Server¹</th>
<th>New Client/Old Server</th>
<th>New Client/New Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCILobArrayRead()</td>
<td>²NA</td>
<td>OK until piece size and offset are &lt; 4 GB.</td>
<td>OK</td>
</tr>
<tr>
<td>OCILobArrayWrite()</td>
<td>NA</td>
<td>OK until piece size and offset are &lt; 4 GB.</td>
<td>OK</td>
</tr>
<tr>
<td>OCILobCopy2()</td>
<td>NA</td>
<td>OK until LOB size, piece size (amount) and offset are &lt; 4 GB.</td>
<td>OK</td>
</tr>
<tr>
<td>OCILobCopy()</td>
<td>OK; limit is 4 GB.</td>
<td>OK</td>
<td>OK; limit is 4 GB.</td>
</tr>
<tr>
<td>OCILobErase2()</td>
<td>NA</td>
<td>OK until piece size and offset are &lt; 4 GB.</td>
<td>OK</td>
</tr>
<tr>
<td>OCILobErase()</td>
<td>OK; limit is 4 GB.</td>
<td>OK</td>
<td>OK; limit is 4 GB.</td>
</tr>
<tr>
<td>OCILobGetLength2()</td>
<td>NA</td>
<td>OK</td>
<td>OK; OCI_ERROR if LOB size &gt; 4 GB.</td>
</tr>
<tr>
<td>OCILobGetLength()</td>
<td>OK; limit is 4 GB.</td>
<td>OK</td>
<td>OK; OCI_ERROR if LOB size &gt; 4 GB.</td>
</tr>
<tr>
<td>OCILobLoadFromFile2()</td>
<td>NA</td>
<td>OK until LOB size, piece size (amount), and offset are &lt; 4 GB.</td>
<td>OK</td>
</tr>
<tr>
<td>OCILobLoadFromFile()</td>
<td>OK; limit is 4 GB.</td>
<td>OK</td>
<td>OK; limit is 4 GB.</td>
</tr>
<tr>
<td>OCILobRead2()</td>
<td>NA</td>
<td>OK until LOB size, piece size (amount), and offset are &lt; 4 GB.</td>
<td>OK</td>
</tr>
</tbody>
</table>
### Table 7–1 (Cont.) LOB Functions Compatibility and Migration

<table>
<thead>
<tr>
<th>LOB Function</th>
<th>Old Client/New or Old Server(^1)</th>
<th>New Client/Old Server</th>
<th>New Client/New Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCILobRead()</td>
<td>OK; limit 4 GB.</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>With new server:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCI_ERROR is returned if you try to read any amount &gt;= 4 GB from any offset &lt; 4 GB. This is because when you read any amount &gt;= 4 GB, that results in an overflow of returned value in <em>amtp</em>, and so it is flagged as an error. Note:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ If you read up to 4 GB – 1 from offset, that is not flagged as an error.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ When you use streaming mode with polling, no error is returned if no attempt is made to use piece size &gt; 4 GB (you can read data &gt; 4 GB in this case).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCILobTrim2()</td>
<td>NA</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>OCILobTrim()</td>
<td>OK; limit 4 GB.</td>
<td>OK</td>
<td>OK; limit 4 GB.</td>
</tr>
<tr>
<td>OCILobWrite2()</td>
<td>NA</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>OCILobWrite()</td>
<td>OK; limit 4 GB.</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>With new server:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCI_ERROR is returned if you write any amount &gt;= 4 GB (from any offset &lt; 4 GB) because that results in an overflow of returned value in <em>amtp</em>. 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.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

**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 CLOBs and NCLOBs, regardless of the client-side character set.

These LOB operations refer to the amount of LOB data on the server. When the client-side character set is of varying width, the following general rules apply to the amount and offset parameters in LOB calls:

- **amount** - When the amount parameter refers to the server-side LOB, the amount is in characters. When the amount parameter refers to the client-side buffer, the amount is in bytes.
- **offset** - Regardless of whether the client-side character set is varying-width, the offset parameter is always in characters for CLOBs or NCLOBs and in bytes for BLOBs or BFILES.

Exceptions to these general rules are noted in the description of the specific LOB call.

**See Also:**
- "LOB Functions" on page 17-17
- "Buffer Expansion During OCI Binding" on page 5-29

**Improving LOB Read/Write Performance**

Here are some hints to improve performance.
Using Data Interface for LOBs
You can bind or define character data for a CLOB column or RAW data for a BLOB column. This requires only one round-trip for inserting or selecting a LOB, as opposed to the traditional LOB interface that requires multiple round-trips.

See Also:
- "Binding LOB Data" on page 5-9 for usage and examples for both INSERT and UPDATE statements
- "Defining LOB Data" on page 5-16 for usage and examples of SELECT statements

Using OCILobGetChunkSize()
OCILobGetChunkSize() returns the usable chunk size in bytes for BLOBs, CLOBs, and NCLOBs. 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.

See Also: ""Options of SecureFiles LOBs" on page 7-22

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.

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.

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

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" on page C-3 for a list of the server round-trips required for each function

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 BFILES, 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 BFILES.

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.

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. See the SESSION_MAX_OPEN_FILES parameter in Oracle Database Reference. Assigning an already opened locator to another locator does not count as opening a new LOB.
LOB and BFILE Operations

- 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:**

- [Appendix B for examples of the use of the OCILobOpen() and OCILobClose() calls in the online demonstration programs](#)
- Table C–2, "Server Round-Trips for OCILob Calls"

### LOB Read and Write Callbacks

OCI supports read and write callback functions. The following sections describe the use of callbacks in more detail.

#### 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.

#### 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 *ctxxp, CONST void *bufp, oraub8 len, ub1 piece,
                        void **changed_bufpp, oraub8 *changed_lenp);
```

The first parameter, `ctxxp`, 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 `ctxxp` 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 `buf1` 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 7–2 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 7–2 Implementing Read Callback Functions Using OCILObRead2()**

```c
...  
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.
  
  report_error();
  
...  
sb4 cbk_read_lob(ctxxp, bufxp, len, piece, changed_bufpp, changed_lenp)
void *ctxxp;
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\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\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;
}

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 7–3 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 7–3 Implementing Write Callback Functions Using OCILobWrite2()

...
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.
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.

```sql
INSERT INTO tab1 VALUES(EMPTY_CLOB())
```

---

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.
Freeing Temporary LOBs

Any time that your OCI program obtains a LOB locator from SQL or PL/SQL, use the \texttt{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 \texttt{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:

\begin{verbatim}
OCILobIsTemporary(env, err, locator, is_temporary);
if(is_temporary)
  OCILobFreeTemporary(svc, err, locator);
\end{verbatim}

\textbf{Take Care When Assigning Pointers}

Special care must be taken when assigning \texttt{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 \texttt{OCILobAssign()} or \texttt{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 \texttt{OCILobFreeTemporary()}. When \texttt{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 \texttt{OCILobFreeTemporary()} call.

\textbf{Temporary LOB Example}

\textit{Example 7–4} shows how temporary LOBs can be used.

\begin{verbatim}
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Note: When a duration is started with \texttt{OCIDurationBegin()}, one of the parameters is the identifier of a parent duration. When a parent duration is ended, all child durations are also ended.

\end{verbatim}
#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 (OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4) strlen((char *)sqlstmt), (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
 { (void) printf("FAILED: OCIStmtPrepare() sqlstmt
");
   return OCI_ERROR;
 }
 /* Define for BLOB */
 if (OCIDefineByPos(stmthp, &defnp1, errhp, (ub4)1, (void *) &lob_loc, (sb4)0, (ub4) OCI_DTYPE_LOB, (size_t)0, (void**)0))
 { (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, 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, OCIHandleAlloc( (void *) envhp, (void **) &errhp, OCI_HTYPE_ERROR,
                                   (size_t)0, (void **)0);
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 BFILES.

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.

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.

Note that the prefetch size is in number of bytes for BLOBs and BFILES and in number of characters for CLOBs.

Example 7–5 shows a code fragment illustrating these steps.

Example 7–5  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, ...);
OCIdefByPos(..., 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,
    (void*) &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, ...);

...  // 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()
Options of SecureFiles LOBs

- 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.

**See Also:**
- "OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE" on page A-18
- "OCI_ATTR_LOBPREFETCH_LENGTH" on page A-38
- "OCI_ATTR_LOBPREFETCH_SIZE" on page A-38

**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."

### Options of SecureFiles LOBs

For SecureFiles (LOBs with the `STORE AS SECUREFILE` option, which were introduced in Oracle Database 11g 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()
Options of SecureFiles LOBs

- 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
Managing Scalable Platforms

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
- Edition-Based Redefinition
- OCI Security Enhancements
- Overview of OCI Multithreaded Development
- OCIThread Package

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.
Levels of Transactional Complexity

OCI supports several levels of transaction complexity, including the following:

- Simple Local Transactions
- Serializable or Read-Only Local Transactions
- Global Transactions

Simple Local Transactions

Many applications work with only simple local transactions. In these applications, an implicit transaction is created when the application makes database changes. The only transaction-specific calls needed by such applications are:

- OCITransCommit() to commit the transaction
- OCITransRollback() to roll back the transaction

As soon as one transaction has been committed or rolled back, the next modification to the database creates a new implicit transaction for the application.

Only one implicit transaction can be active at any time on a service context. Attributes of the implicit transaction are opaque to the user.

If an application creates multiple sessions, each one can have an implicit transaction associated with it.

See Also: "OCITransCommit()" on page 17-148 for sample code showing the use of simple local transactions.

Serializable or Read-Only Local Transactions

Applications requiring serializable or read-only transactions require an additional OCI OCITransStart() call to start the transaction.

The OCITransStart() call must specify OCI_TRANS_SERIALIZABLE or OCI_TRANS_READONLY, as appropriate, for the flags parameter. If no flag is specified, the default value is OCI_TRANS_READWRITE for a standard read/write transaction.

Specifying the read-only option in the OCITransStart() call saves the application from performing a server round-trip to execute a SET TRANSACTION READ ONLY statement.

Global Transactions

Global transactions are necessary only in more sophisticated transaction-processing applications.

Transaction Identifiers

Three-tier applications such as transaction processing (TP) monitors create and manage global transactions. They supply a global transaction identifier (XID) that a server 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 8–1 provides the structure for one possible XID of 1234.

<table>
<thead>
<tr>
<th>Component</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtrid</td>
<td>12</td>
</tr>
<tr>
<td>bqual</td>
<td>34</td>
</tr>
<tr>
<td>gtrid+bqual=XID</td>
<td>1234</td>
</tr>
</tbody>
</table>

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.

**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.

**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 8–1 illustrates tightly coupled branches within an application. The XIDs 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.
Branch States
Transaction branches are classified into two states: active branches and inactive branches.

A branch is active if a server process is executing requests on the branch. A branch is inactive if no server processes are executing requests in the branch. In this case, no session is the parent of the branch, and the branch becomes owned by the PMON process in the server.

Detaching and Resuming Branches
A branch becomes inactive when an OCI application detaches it, using the OCITransDetach() call. The branch can be made active again by resuming it with a call to OCITransStart() with the flags parameter set to OCI_TRANS_RESUME.

See Also: "OCITransStart()" on page 17-156 for sample code demonstrating this scenario.
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.

### 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 `OCIArrAttrSet()` before logging on and using global transactions.

### 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.
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.

Transaction Examples

Table 8–1 through Table 8–5 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.

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.

Update Successfully, One-Phase Commit

Table 8–2 lists the steps for a one-phase commit operation.

<table>
<thead>
<tr>
<th>Step</th>
<th>OCI Action</th>
<th>XID</th>
<th>Flags</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OCITransStart()</td>
<td>1234</td>
<td>OCI_TRANS_NEW</td>
<td>Starts new read/write transaction</td>
</tr>
<tr>
<td>2</td>
<td>SQL UPDATE</td>
<td>-</td>
<td>-</td>
<td>Update rows</td>
</tr>
<tr>
<td>3</td>
<td>OCITransCommit()</td>
<td>-</td>
<td>-</td>
<td>Commit succeeds.</td>
</tr>
</tbody>
</table>

Start a Transaction, Detach, Resume, Prepare, Two-Phase Commit

Table 8–3 lists the steps for a two-phase commit operation.
In Step 4, the transaction can be resumed by a different process, as long as it had the same authorization.

### Table 8–4 Read-Only Update Fails

Table 8–4 lists the steps in a failed read-only update operation.

<table>
<thead>
<tr>
<th>Step</th>
<th>OCI Action</th>
<th>XID</th>
<th>Flags</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OCITransStart()</td>
<td>1234</td>
<td>OCI_TRANS_NEW</td>
<td>Starts new read-only transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OCI_TRANS_READONLY</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SQL UPDATE</td>
<td>-</td>
<td>-</td>
<td>Update fails, because the transaction is read-only.</td>
</tr>
<tr>
<td>3</td>
<td>OCITransCommit()</td>
<td>-</td>
<td>-</td>
<td>Commit has no effect.</td>
</tr>
</tbody>
</table>

### Table 8–5 Read-Only Transaction

Table 8–5 lists the steps for a read-only transaction.

<table>
<thead>
<tr>
<th>Step</th>
<th>OCI Action</th>
<th>XID</th>
<th>Flags</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OCITransStart()</td>
<td>1234</td>
<td>OCI_TRANS_NEW</td>
<td>Starts new read-only transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OCI_TRANS_READONLY</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SQL SELECT</td>
<td>-</td>
<td>-</td>
<td>Queries the database</td>
</tr>
<tr>
<td>3</td>
<td>OCITransCommit()</td>
<td>-</td>
<td>-</td>
<td>No effect — transaction is read-only, no changes made</td>
</tr>
</tbody>
</table>

### Password and Session Management

OCI can authenticate and maintain multiple users.

### OCI Authentication Management

The OCISessionBegin() call authenticates a user against the server set in the service context handle. It must be the first call for any given server handle. OCISessionBegin() authenticates the user for access to the Oracle 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`, and `OCI_PRELIM_AUTH` settings can only be used with a primary user session context.

A migratable session can be switched, or migrated, to a server handle within an environment represented by an environment handle. It can also migrate or be cloned to a server handle in another environment in the same process, or in a different process in a different mode. To perform this migration or cloning, you 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.

   **See Also:** "User Session Handle Attributes" on page A-14

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.
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.

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

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 8–1.

**Example 8–1  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);
```
The server group name is an alphanumeric string not exceeding 30 characters. This attribute can only be set after calling `OCIHandlerAttach()`. `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.

**See Also:** "Server Handle Attributes" on page A-11

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`.

### 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.

**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.

**OCI_CRED_PROXY**

Use **OCI_CRED_PROXY** as the value passed in the **credt** parameter of **OCISessionBegin()** when an application server starts a session on behalf of a client, rather than **OCI_CRED_RDBMS** (database user name and password required) or **OCI_CRED_EXT** (externally provided credentials).

**OCI_ATTR_PROXY_CREDENTIALS**

Use 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 8–2.

**Example 8–2 Defining the OCI_ATTR_PROXY_CREDENTIALS Attribute to Specify the Credentials of the Application Server for Client Authentication**

```c
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);
```

**OCI_ATTR_DISTINGUISHED_NAME**

Your applications can use the distinguished name contained within a X.509 certificate as the login name of the client, instead of the database user name.

To pass the distinguished name of the client, the middle-tier server calls **OCIAttrSet()**, passing **OCI_ATTR_DISTINGUISHED_NAME**, as shown in Example 8–3.

**Example 8–3 Defining the OCI_ATTR_DISTINGUISHED_NAME Attribute to Pass the Distinguished Name of the Client**

```c
/* Declarations */
...
OCIAttrSet((void *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (void *)distinguished_name, (ub4) 0,
           OCI_ATTR_DISTINGUISHED_NAME, error_handle);
```
OCI_ATTR_CERTIFICATE
Certificate-based proxy authentication using OCI_ATTR_CERTIFICATE will not be supported in future Oracle Database releases. Use OCI_ATTR_DISTINGUISHED_NAME or OCI_ATTR_USERNAME attribute instead. This method of authentication is similar to the use of 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 8–4.

Example 8–4  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);

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 8–5.

Example 8–5  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);

OCI_ATTR_CLIENT_IDENTIFIER
Many middle-tier applications connect to the database as an application, and rely on the middle tier to track end-user identity. To integrate tracking of 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 8–6. On the next request to the server, the information is propagated and stored in the server session.

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.

Example 8–6  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);

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:
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 8–7.

Example 8–7  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);

See Also:  "User Session Handle Attributes" on page A-14

Example 8–8 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 8–8  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) OCIIHandleAlloc( (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) OCIHandleAlloc((void *) environment_handle, (void **) &application_server_session_handle, 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));

(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);

(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);

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));

*/
(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
*/
...

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.
Externally Initialized Context Attributes in OCI

The client applications you develop can set application contexts explicitly in the session handle before authentication, using the following attributes in OCI functions:

**OCI_ATTR_APPCTX_SIZE**
Use the **OCI_ATTR_APPCTX_SIZE** attribute to initialize the context array size with the desired number of context attributes in the OCIAttrSet() call, as shown in Example 8–9.

Example 8–9  Defining the OCI_ATTR_APPCTX_SIZE Attribute to Initialize the Context Array Size with the Desired Number of Context Attributes

```c
OCIAttrSet(session, (ub4) OCI_HTYPE_SESSION, (void *)&size, (ub4)0, OCI_ATTR_APPCTX_SIZE, error_handle);
```

**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, as shown in Example 8–10. (The parameter `ctxl_desc` must be of data type `OCIParam *`).

Example 8–10  Using the OCI_ATTR_APPCTX_LIST Attribute to Get a Handle on the Application Context List Descriptor for the Session

```c
OCIAttrGet(session, (ub4) OCI_HTYPE_SESSION, (void *)&ctxl_desc, (ub4)0, OCI_ATTR_APPCTX_LIST, error_handle);
```

Example 8–11 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 8–11  Calling OCIParamGet() to Obtain an Individual Descriptor for the i-th Application Context Using the Application Context List Descriptor

```c
OCIParamGet(ctxl_desc, OCI_DTYPE_PARAM, error_handle,(void **)&ctx_desc, i);
```

### Session Handle Attributes Used to Set an Externally Initialized Context

Set the appropriate values of the application context using these attributes:

- **OCI_ATTR_APPCTX_NAME** to set the namespace of the context, which must be a valid SQL identifier.
- **OCI_ATTR_APPCTX_ATTR** to set an attribute name in the given context, a non-case-sensitive string of up to 30 bytes.
- **OCI_ATTR_APPCTX_VALUE** to set the value of an attribute in the given context.

Each namespace can have many attributes, each of which has one value. Example 8–12 shows the calls you can use to set them.

**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
Example 8–12  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);

Note that only character type is supported, because application context operations are based on the VARCHAR2 data type.

See Also:  "User Session Handle Attributes" on page A-14

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.

OCI_ATTR_COLLECT_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.

OCI_ATTR_CALL_TIME

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 8–13 shows how to do this in a code fragment.

Example 8–13  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);
...

Attributes for End-to-End Application Tracing

Set these attributes for tracing and debugging applications:

- OCI_ATTR_MODULE - Name of the current module in the client application.
- OCI_ATTR_ACTION - Name of the current action within the current module. Set to NULL if you do not want to specify an action.
- OCI_ATTR_CLIENT_INFO - Client application additional information.
Using OCI::SessionBegin() with an Externally Initialized Context

When you call OCI::SessionBegin(), the context set in the session handle is pushed to
the server. No additional contexts are propagated to the server session. Example 8–14
illustrates the use of these calls and attributes.

Example 8–14 Using OCI::SessionBegin() with an Externally Initialized Context

```c
#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,
    (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) OCIHandleAlloc( (void *) envhp, (void **) &srvhp, OCI_HTYPE_SVCCTX,
    (size_t) 0, (void **) 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);
```

See Also: "User Session Handle Attributes" on page A-14
/** 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);
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, \texttt{nm}, is created:

\begin{verbatim}
CREATE CONTEXT \texttt{nm} USING hr.package1;
\end{verbatim}

To modify the data grouped in that namespace, users must execute the designated PL/SQL package, \texttt{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 \texttt{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 \texttt{nm} namespace, then it could set it to \texttt{NULL} or an empty string. This is referred to as the clear operation in this document.

To clear all information in the \texttt{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.

Multiple SET Operations

Use the \texttt{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 8–15, 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.

\begin{verbatim}
Example 8–15 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);
\end{verbatim}
Client Application Context

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 8–16 using the OCIAppCtxSet() function.
Example 8–16

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);

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 8–17.
Example 8–17

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);

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.

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

Managing Scalable Platforms 8-21


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:
- "OCIAppCtxClearAll()" on page 16-4
- "OCIAppCtxSet()" on page 16-5

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 8–18.

Example 8–18 Calling OCIAttrSet() to Set the OCI_ATTR_EDITION Attribute

```c
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:
- "OCIAppCtxClearAll()" on page 16-4
- "OCIAppCtxSet()" on page 16-5
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.

**See Also:**
- "OCI_ATTR_EDITION" on page A-19
- *Oracle Database Advanced Application Developer's Guide* for a more complete description of edition-based redefinition
- "Restrictions on Attributes Supported for OCI Session Pools" on page 16-38

**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), and 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*, section about embedding calls in middle-tier applications to get, set, and clear client session IDs

**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:

```sql
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.

**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 `OCIHandleGet()`.

**See Also:** "OCI_ATTR_ACCESS_BANNER" on page A-11

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()`, `OCIHandleGet()`, `OCILogon()`, or `OCILogon2()`.

**See Also:** "OCI_ATTR_AUDIT_BANNER" on page A-16

**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.

**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.
Advantages of OCI Thread Safety

The implementation of thread safety in OCI has the following advantages:

- Multiple threads of execution can make OCI calls with the same result as successive calls made by a single thread.
- When multiple threads make OCI calls, there are no side effects between threads.
- Users who do not write multithreaded programs do not pay a performance penalty for using thread-safe OCI calls.
- Use of multiple threads can improve program performance. Gains may be seen on multiprocessor systems where threads run concurrently on separate processors, and on single processor systems where overlap can occur between slower operations and faster operations.

OCI Thread Safety and Three-Tier Architectures

In addition to client/server applications, where the client can be a multithreaded program, a typical use of multithreaded applications is in three-tier (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.

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.

---

**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" on page 2-27

**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.

---

**OCIThread Package**

The OCIThread package provides some commonly used threading primitives. It offers a portable interface to threading capabilities native to various operating systems, but does not implement threading on operating systems that do not have native threading capability.

OCIThread does not provide a portable implementation, but it serves as a set of portable covers for native multithreaded facilities. Therefore, operating systems that do not have native support for multithreading 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 OCIThreadIsMulti() call.

See Also:
■ "Thread Management Functions" on page 17-121
■ cdemothr.c in the demo directory is an example of a multithreading application

Initialization and Termination

The types and functions described in this section are associated with the initialization and termination of the OCIThread package. OCIThread must be initialized before 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 8–6 lists functions for thread initialization and termination.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIThreadProcessInit()</td>
<td>Performs OCIThread process initialization</td>
</tr>
<tr>
<td>OCIThreadInit()</td>
<td>Initializes OCIThread context</td>
</tr>
<tr>
<td>OCIThreadTerm()</td>
<td>Terminates the OCIThread layer and frees context memory</td>
</tr>
<tr>
<td>OCIThreadIsMulti()</td>
<td>Tells the caller whether the application is running in a multithreaded environment or a single-threaded environment</td>
</tr>
</tbody>
</table>

See Also: "Thread Management Functions" on page 17-121
OCIThread Context
Most calls to OCIThread functions use the OCI environment or user session handle as a parameter. The OCIThread context is part of the OCI environment or user session handle, and it must be initialized by calling OCIThreadInit(). Termination of the OCIThread context occurs by calling OCIThreadTerm().

---

**Note:** The OCIThread context is an opaque data structure. Do not attempt to examine the contents of the context.

---

Passive Threading Primitives
The passive threading primitives deal with the manipulation of mutex, thread 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 8–7 lists functions used to implement passive threading.

**Table 8–7 Passive Threading Primitives**

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIThreadMutexInit()</td>
<td>Allocates and initializes a mutex</td>
</tr>
<tr>
<td>OCIThreadMutexDestroy()</td>
<td>Destroys and deallocates a mutex</td>
</tr>
<tr>
<td>OCIThreadMutexAcquire()</td>
<td>Acquires a mutex for the thread in which it is called</td>
</tr>
<tr>
<td>OCIThreadMutexRelease()</td>
<td>Releases a mutex</td>
</tr>
<tr>
<td>OCIThreadKeyIdInit()</td>
<td>Allocates and generates a new key</td>
</tr>
<tr>
<td>OCIThreadKeyIdDestroy()</td>
<td>Destroys and deallocates a key</td>
</tr>
<tr>
<td>OCIThreadKeyIdGet()</td>
<td>Gets the calling thread’s current value for a key</td>
</tr>
<tr>
<td>OCIThreadKeyIdSet()</td>
<td>Sets the calling thread’s value for a key</td>
</tr>
<tr>
<td>OCIThreadKeyIdInit()</td>
<td>Allocates and initializes a thread ID</td>
</tr>
<tr>
<td>OCIThreadKeyIdDestroy()</td>
<td>Destroys and deallocates a thread ID</td>
</tr>
<tr>
<td>OCIThreadKeyIdSet()</td>
<td>Sets one thread ID to another</td>
</tr>
<tr>
<td>OCIThreadKeyIdSetNull()</td>
<td>Nulls a thread ID</td>
</tr>
<tr>
<td>OCIThreadKeyIdGet()</td>
<td>Retrieves a thread ID for the thread in which it is called</td>
</tr>
<tr>
<td>OCIThreadKeyIdSame()</td>
<td>Determines if two thread IDs represent the same thread</td>
</tr>
<tr>
<td>OCIThreadKeyIdNull()</td>
<td>Determines if a thread ID is NULL</td>
</tr>
</tbody>
</table>

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()`.

**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.

**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.

**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.
Active Threading Primitives

The active threading primitives deal with manipulation of actual threads. Because specifications of most of these primitives require multiple threads, they work correctly only in the enabled OCIThread. In the disabled OCIThread, they always return an error. The exception is OCIThreadHandleGet(); it may be called in a single-threaded environment and has no effect.

Active primitives can only be called by code running in a multithreaded environment. You can call OCIThreadIsMulti() to determine whether the environment is multithreaded or single-threaded. Table 8–8 lists functions used to implement active threading.

Table 8–8  Active Threading Primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIThreadHndInit()</td>
<td>Allocates and initializes a thread handle</td>
</tr>
<tr>
<td>OCIThreadHndDestroy()</td>
<td>Destroys and deallocates a thread handle</td>
</tr>
<tr>
<td>OCIThreadCreate()</td>
<td>Creates a new thread</td>
</tr>
<tr>
<td>OCIThreadJoin()</td>
<td>Allows the calling thread to join with another</td>
</tr>
<tr>
<td>OCIThreadClose()</td>
<td>Closes a thread handle</td>
</tr>
<tr>
<td>OCIThreadHandleGet()</td>
<td>Retrieves a thread handle</td>
</tr>
</tbody>
</table>

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().
This chapter contains these topics:

- Connection Pooling in OCI
- Session Pooling in OCI
- Database Resident Connection Pooling
- When to Use Connection Pooling, Session Pooling, or Neither
- Statement Caching in OCI
- User-Defined Callback Functions in OCI
- Transparent Application Failover in OCI
- HA Event Notification
- OCI and Streams Advanced Queuing
- Publish-Subscribe Notification in OCI

**Connection Pooling in OCI**

This section includes the following topics:

- OCI Connection Pooling Concepts
- OCI Calls for Connection Pooling
- Examples of OCI Connection Pooling

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.

**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
Connection pooling in OCI 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.

**Similarities and Differences from a Shared Server**

Connection pooling on the middle tier 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 middle tier 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.

**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. If you must work with stateless sessions, see "Session Pooling in OCI" on page 9-7.
**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 9–1 illustrates OCI connection pooling.

**Figure 9–1  OCI Connection Pooling**

![Diagram of OCI Connection Pooling](image)

**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

**OCI Calls for Connection Pooling**

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
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:

```c
OCICPool *poolhp;
OCIHandleAlloc((void *) envhp, (void **) &poolhp, OCI_HTYPE_CPOOL,
(size_t) 0, (void **) 0));
```

Create the Connection Pool

The function OCIConnectionPoolCreate() initializes the connection pool handle. It has these IN parameters:

- `connMin`, the minimum number of connections to be opened when the pool is created.
- `connIncr`, the incremental number of connections to be opened when all the connections are busy and a call needs a connection. This increment is used only when the total number of open connections is less than the maximum number of connections that can be opened in that pool.
- `connMax`, the maximum number of connections that can be opened in the pool. When the maximum number of connections are open in the pool, and all the connections are busy, if a call needs a connection, it 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:

```c
OCIConnectionPoolCreate((OCIEnv *)envhp,
(OCIError *)errhp, (OCICPool *)poolhp,
&poolName, &poolNameLen,
(text *)database,strlen(database),
```
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 OCIVectorizerAttach() 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.

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.

See Also: Oracle Database Performance Tuning Guide, the section about configuring a shared server

If you are still running into the SGA limitation, you must consider:

- Reducing the session memory consumption by having fewer open statements for each session
- Reducing the number of sessions in the back end by pooling sessions on the mid-tier
- Or otherwise, turning off connection pooling

The application must avoid using dedicated database links on the back end with connection pooling.

If the back end is a dedicated server, effective connection pooling 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 Administrator's Guide, the section on shared database links for more information about distributed databases

Log Off from the Database
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.
Session Pooling in OCI

- **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.

**Destroy the Connection Pool**
Use OCIConnectionPoolDestroy() to destroy the connection pool.

**Free the Pool Handle**
The pool handle is freed using OCIHandleFree().

These last three actions are illustrated in this code fragment:

```c
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:**
- "Connection Pool Handle Attributes" on page A-22
- "OCIConnectionPoolCreate()" on page 16-7, "OCILogon2()" on page 16-24, and "OCIConnectionPoolDestroy()" on page 16-9

**Examples of OCI Connection Pooling**
Examples of connection pooling in tested complete programs can be found in cdemocp.c and cdemocpproxy.c in directory demo.

**Session Pooling in OCI**
This section includes the following topics:

- Functionality of OCI Session Pooling
- Homogeneous and Heterogeneous Session Pools
- Using Tags in Session Pools
- OCI Handles for Session Pooling
- Using OCI Session Pooling
- OCI Calls for Session Pooling
- Example of OCI Session Pooling
- Runtime Connection Load Balancing
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.

Functionality of OCI Session Pooling

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.

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.

Using Tags in Session Pools

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 NLS settings), and return the session to the pool, labeling it with an appropriate tag in
the OCISessionRelease() call.

The user, or some other user, 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.

See Also: "OCISessionGet()" on page 16-34 for a further discussion of tagging sessions

OCI Handles for Session Pooling

The following handle types are for session pooling.

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.

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. See User Session Handle Attributes for more information. The authentication information handle has the attribute type OCI_HTYPE_AUTHINFO (see Table 2–1).

An example of the OCIHandleAlloc() call follows:

OCIAuthInfo *authp;
OCIHandleAlloc((void *) envhp, (void **) &authp, OCI_HTYPE_AUTHINFO,
(size_t) 0, (void **) 0));

See Also:

- "User Session Handle Attributes" on page A-14 for the attributes that belong to the authentication information handle
- "Session Pool Handle Attributes" on page A-24 for more information about the session pooling attributes
- "Connect, Authorize, and Initialize Functions" on page 16-3 for complete information about the functions used in session pooling
- See “OCISessionGet()” on page 16-34 for details of the session handle attributes that you can use with this call

Using OCI Session Pooling

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 `OCIStmPrepare2()` 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 `OCI_SESSION_RELEASE`
   j. Frees the authentication information handle with `OCIHandleFree()`
   k. Ends the loop for each thread

4. Destroy the session pool using `OCISessionPoolDestroy()`.

**OCI Calls for Session Pooling**

Here are 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
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:

```c
OCISpool *poolhp;
OCIHandleAlloc((void *) envhp, (void **) &poolhp, OCI_HTYPE_SPOOL, (size_t) 0,
              (void **) 0));
```

Create the Pool Session
You can use the function OCI_SessionPoolCreate() to create the session pool. Here is an example of how to use this call:

```c
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);
```

Log On to the Database
You can use these calls to log on to the database in session pooling mode.

- **OCILoginForm2()**
  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 OCILoginForm2() to log on to the database in session pooling mode:
  ```c
  for (i = 0; i < MAXTHREADS; ++i)
  {
    OCILoginForm2(envhp, errhp, &svchp[i], "hr", 2, "hr", 2, poolName,
                poolNameLen, OCI_LOGON2_SPOOL));
  }
  ```

- **OCI_SessionGet()**
  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 OCI_SessionGet() follows. It is taken from cdemo3sp.c in the demo directory.
  ```c
  OCI_SessionGet(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 OCI_SessionGet() (or OCILoginForm2()), 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.
Log Off from the Database

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.

Destroy the Session Pool

Call **OCISessionPoolDestroy()** to destroy the session pool, as shown in the following example:

```c
OCISessionPoolDestroy(poolhp, errhp, OCI_DEFAULT);
```

Free the Pool Handle

Call **OCIHandleFree()** to free the session pool handle, as shown in the following example:

```c
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.

---

Example of OCI Session Pooling

For an example of session pooling in a tested complete program, see `cdemosp.c` in directory demo.

Runtime Connection Load Balancing

Oracle Real Application Clusters (Oracle RAC) is a database option in which a single database is hosted by multiple instances on multiple nodes. The Oracle RAC shared disk method of clustering databases increases scalability. The nodes can easily be added or freed to meet current needs and improve availability, because if one node fails, another can assume its workload. Oracle RAC adds high availability and failover capacity to the database, because all instances have access to the whole database.

Balancing of work requests occurs at two different times: at connect time and at runtime. These are referred to as **connect time load balancing** (provided by Oracle Net Services) and **runtime connection load balancing**. For Oracle RAC environments, session pools use service metrics received from the Oracle RAC load balancing advisory through Fast Application Notification (FAN) events to balance application session requests. The work requests coming into the session pool can be distributed across the instances of Oracle RAC offering a service, using the current service performance.
Connect time load balancing occurs when a session is first created by the application. It is necessary that the sessions that are part of the pool be well distributed across Oracle RAC instances, when they are first created. This ensures that sessions on each of the instances get a chance to execute work.

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.

Runtime connection load balancing is enabled by default in an Oracle Database Release 11.1 or later client communicating with a server of Oracle Database Release 10.2 or later. Setting the mode parameter to OCI_SPC_NO_RLB when calling OCISessionPoolCreate() disables runtime connection load balancing.

Receiving Load Balancing Advisory FAN Events
To receive the service metrics based on the service time, the following requirements must be met:

- Oracle RAC environment with Oracle Clusterware must be set up and enabled.
- The application must have been linked with the threads library.
- The OCI environment must be created in OCI_EVENTS and OCI_THREADED mode.
- The service must be modified to set up its goal and the connection load balancing goal as follows:

  ```sql
  EXEC DBMS_SERVICE.MODIFY_SERVICE('myService',
    DBMS_SERVICE.GOAL_SERVICE_TIME,
    clb_goal => dbms_service.clb_goal_short);
  ```

See Also:

- "OCISessionPoolCreate()" on page 16-40
- Oracle Real Application Clusters Administration and Deployment Guide, the section about enabling OCI clients to receive FAN events
- Oracle Database PL/SQL Packages and Types Reference, "DBMS_SERVICE"

Database Resident Connection Pooling
This section includes the following topics:

- Configuring Database Resident Connection Pooling
- Using OCI Session Pool APIs with DRCP
- Session Purity and Connection Class
- Starting the Database Resident Connection Pool
- Enabling 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, which are referred to as pooled servers. Pooled servers can be shared across multiple applications running on the same or several hosts. A connection broker process manages the pooled servers at the database instance level. DRCP is a configurable feature chosen at program runtime, allowing traditional and DRCP-based connection architectures to be in concurrent use.

DRCP is especially relevant for architectures with multiprocess single-threaded application servers (such as PHP and Apache) that cannot do middle-tier connection pooling. DRCP is also very useful in large scale Web deployments where hundreds or thousands of web servers or mid-tiers need database access, client-side pools (even in multithreaded systems and languages such as Java). Using DRCP, the database can scale to tens of thousands of simultaneous connections. If your Database web application must scale with large numbers of connections, DRCP is your connection pooling solution.

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, across multiple middle-tier hosts, and across multiple middle-tiers (web servers, containers) accommodating applications written in different languages. 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.

Clients get connections out of the database resident connection pool connect to an Oracle Database background process known as the connection broker. The connection broker implements the pool functionality and multiplexes pooled servers among persistent inbound connections from the client.

When a client requires database access, the connection broker picks up a server process from the pool and hands it off to the client. The client is then directly connected to the server process until the request is served. After the server has finished, the server process is released back into the pool and the connection from the client is restored to the connection broker as a persistent inbound connection from the client process. In DRCP, releasing resources leaves the session intact, but no longer associated with a connection (server process). Because this session stores its user global area (UGA) in the program global area (PGA), not in the system global area (SGA), a client can reestablish a connection transparently upon detecting activity.
DRCP is typically preferred for applications with a large number of connections. Shared servers are useful for a medium number of connections and dedicated sessions are preferred for small numbers of connections. The threshold sizes are relative to the amount of memory available on the database host.

DRCP provides the following advantages:

- It enables resource sharing among multiple client applications and middle-tier application servers.
- It improves scalability of databases and applications by reducing resource usage on the database host.

Compared to shared servers, DRCP offers these additional benefits:

- DRCP provides a direct tie with the database server furnished by client-side connection pooling (that is, there is no man-in-the-middle like client-side connection pooling, but unlike shared servers)
- DRCP can pool database servers (like client-side connection pooling and shared servers),
- DRCP can pool sessions (like client-side connection pooling and unlike shared servers)
- DRCP can share connections across mid-tier boundaries (unlike client-side connection pooling)

DRCP offers a unique connection pooling solution that addresses scalability requirements in environments requiring large numbers of connections with minimal database resource usage.

See Also:

- Oracle Database Concepts for details about the DRCP architecture

Configuring Database Resident Connection Pooling

The pool is managed by the DBA using the DBMS_CONNECTION_POOL package. The pool is installed by default, but is shutdown. The DBA must start it and specify DRCP configuration options that include, for example, the minimum and maximum number of pooled servers to be allowed in the pool, the number of connection brokers to be created, and the maximum number of connections that each connection broker can handle, and so forth. See the references for more information.

OCI session pool APIs have been extended to interoperate with the database resident connection pool. See the references for more information.

See Also:

- Oracle Database PL/SQL Packages and Types Reference for details about the DBMS_CONNECTION_POOL package
- Oracle Database Administrator’s Guide for the details of configuring a database for DRCP

Using OCI Session Pool APIs with DRCP

The sections that follow describe OCI session pool APIs that have been extended to interoperate with the database resident connection pool. An OCI application typically initializes the environment for the OCI session pool for DRCP using OCISessionPoolCreate() by specifying the database connection string (connStr),
whether a user name (userid) and password (password) are associated with each session, the minimum (sessMin) and the next increment (sessIncr) of sessions to be started if the mode parameter is specified as OCI_SPC_HOMOGENEOUS to allow all sessions in the pool to be authenticated with the user name and password passed in, the maximum (sessMax) number of sessions allowed in the session pool, and so forth.

Sessions are obtained from DRCP from the OCI session pool using OCISessionGet(), by specifying the OCI_SESSGET_SPOOL attribute in the mode parameter and sessions are released to DRCP to the OCI session pool using OCISessionRelease(). The OCI session pool can also transparently keep connections to the connection broker cached to improve performance. OCI applications can reuse the sessions within which the application leaves sessions of a similar state by using OCISessionGet() and setting the OCI_ATTR_CONNECTION_CLASS attribute and specifying a connection class name or by using the OCIAuthInfo handle before calling OCISessionGet(). Using OCISessionGet() (mode), OCI applications can also specify session purity, that is, whether to reuse a pooled session (set the OCI_SESSGET_PURITY_SELF attribute) or to use a new session (set the OCI_SESSGET_PURITY_NEW attribute).

In addition, features offered by the traditional client-side OCI session pool, such as tagging, statement caching, and transparent application failover (TAF) are also supported with DRCP.

### Session Purity and Connection Class

In Oracle Database Release 11.1, OCI introduced two settings that can be specified when obtaining a session using OCISessionGet():

- **Session Purity**
- **Connection Class**
- **Defaults for Session Purity and Connection Class**

### Session Purity

Session purity specifies whether the application logic is set up to reuse a pooled session or to use a new session. OCISessionGet() has been enhanced to take in a purity setting of OCI_SESSGET_PURITY_NEW or OCI_SESSGET_PURITY_SELF. Alternatively, you can set OCI_ATTR_PURITY_NEW or OCI_ATTR_PURITY_SELF on the OCIAuthInfo handle before calling OCISessionGet(). Both methods are equivalent.

**Note:** When reusing a session from the pool, the NLS attributes of the server take precedence over those of the client.

For example, if the client has NLS_LANG set to french_france.us7ascii and if it is assigned a German session from the pool, the client session would be German.

You can use connection classes to restrict sharing and to avoid this problem.

Example 9–1 shows how a connection pooling application sets up a NEW session.

**Example 9–1 Setting Session Purity**

```c
/* OCIAttrSet method */

ub4 purity = OCI_ATTR_PURITY_NEW;
OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO, &purity, sizeof (purity),
           OCI_ATTR_PURITY, errhp);
```
Connection Class

*Connection class* defines a logical name for the type of connection required by the application. Sessions from the OCI session pool cannot be shared by different users (a session first created for user HR is only given out to subsequent requests by user HR.) The connection class setting allows for further separation between the sessions of a given user. The connection class setting lets different applications (connecting as the same database user) identify their sessions using a logical name that corresponds to the application. OCI then ensures that such sessions belonging to a particular connection class are not shared outside of the connection class.

You can use the *OCI_ATTR_CONNECTION_CLASS* attribute on the OCIAuthInfo handle to set the connection class. The connection class is a string attribute. OCI supports a maximum connection class length of 1024 bytes. The asterisk (*) is a special character and is not allowed in the connection class name.

**Example 9–2** shows that an HRMS application needs sessions identified with the connection class *HRMS*.

**Example 9–2 Setting the Connection Class as HRMS**

```c
OCISessionPoolCreate (envhp, errhp, &spoolhp, &poolName, &poolNameLen, "HRDB", strlen("HRDB"), 0, 10, 1, "HR", strlen("HR"), "HR", strlen("HR"), OCI_SPC_HOMOGENEOUS);

OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO, "HRMS", strlen("HRMS"), OCI_ATTR_CONNECTION_CLASS, errhp);
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0, NULL, NULL, NULL, OCI_SESSGET_SPOOL);
```

**Example 9–3** shows that a recruitment application needs sessions identified with the connection class *RECMS*.

**Example 9–3 Setting the Connection Class as RECMS**

```c
OCISessionPoolCreate (envhp, errhp, &spoolhp, &poolName, &poolNameLen, "HRDB", strlen("HRDB"), 0, 10, 1, "HR", strlen("HR"), "HR", strlen("HR"), OCI_SPC_HOMOGENEOUS);

OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO, "RECMS", strlen("RECMS"), OCI_ATTR_CONNECTION_CLASS, errhp);
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0, NULL, NULL, NULL, OCI_SESSGET_SPOOL);
```

### Defaults for Session Purity and Connection Class

Table 9–1 illustrates the defaults used in various client scenarios.
Database Resident Connection Pooling

### Starting the Database Resident Connection Pool

The database administrator (DBA) must log on as **SYSDBA** and start the default pool, **SYS_DEFAULT_CONNECTION_POOL**, using **DBMS_CONNECTION_POOL.START_POOL** with the default settings.

For detailed information about configuring the pool, see *Oracle Database Administrator’s Guide*.

### Enabling Database Resident Connection Pooling

Any application can benefit from database resident connection pool by specifying **:POOLED** in the Easy Connect string (see Example 9–4) or by specifying **(SERVER=POOLED)** in the TNS connect string (see Example 9–5).

#### Example 9–4  Specifying :POOLED in the Easy Connect String for Enabling DRCP

```
oraclehost.company.com:1521/books.company.com:POOLED
```

#### Example 9–5  Specifying SERVER=POOLED in a TNS Connect String for Enabling DRCP

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=oraclehost.company.com)
(PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)(SERVER=POOLED)))
```

### Benefiting from the Scalability of DRCP in an OCI Application

Consider the following three types of application scenarios and note how each benefits from DRCP:

- Applications that do not use OCI session pooling and also do not specify any connection class or purity setting (or specify a purity setting of **NEW**) get a new session from the DRCP. Similarly, when the application releases a connection back to the pool, the session is not shared with other instances of the same application by default. SQL*Plus is an example of a client that does not use OCI session pooling. It holds on to connections even when the connection is idle. As result, the pool server remains assigned to the client if the client session exists or if the client session does not log off. The application, however, does get the benefit of reusing an existing pooled server process.

- Applications that use the **OCIResultSet** call outside of an OCI session pool, or to specify the connection class and set purity=SELF can reuse both DRCP pooled server processes and sessions. However, following an **OCIResultSet** call, OCI terminates the connection to the connection broker. On the next
OCISessionGet() call, the application reconnects to the broker. Once it reconnects, the DRCP assigns a pooled server (and session) belonging to the connection class specified. Reconnecting, however, incurs the cost of connection establishment and reauthentication. Such applications achieve better sharing of DRCP resources (processes and sessions) but do not get the benefit of caching connections to the connection broker.

- Applications that use OCI session pool APIs and specify the connection class and set `purity=SELF` make full use of the DRCP functionality through reuse of both the pooled server process and the associated session. They get the benefit of cached connections to the connection broker. Cached connections do not incur the cost of reauthentication on the OCISessionGet() call.

**Best Practices for Using DRCP**

The steps to design an application that can benefit from the full power of DRCP are very similar to the steps required for an application that uses the OCI session pool as described in "Using OCI Session Pooling" on page 9-9 and "OCI Calls for Session Pooling" on page 9-10.

The only additional step is that for best performance, when deployed to run with DRCP, the application should specify an explicit connection class setting. Multiple instances of the same application should specify the same connection class setting for best performance and enhanced sharing of DRCP resources. Ensure that the different instances of the application can share database sessions.

Example 9–6 shows a database resident connection pooling DRCP application.

**Example 9–6 Database Resident Connection Pooling Application**

```c
/* Assume that all necessary handles are allocated. */

/* This middletier uses a single database user. Create a homogeneous client-side session pool */
OCISessionPoolCreate(envhp, errhp, spoolhp, &poolName, &poolNameLen, "BOOKSDB",
        strlen("BOOKSDB"), 0, 10, 1, "SCOTT", strlen("SCOTT"), "password",
        strlen("password"), OCI_SPC_HOMOGENEOUS);

while (1)
{
    /* Process a client request */
    WaitForClientRequest();
    /* Application function */

    /* Set the Connection Class on the OCIAuthInfo handle that is passed as argument to OCISessionGet*/
    OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO,  "BOOKSTORE", strlen("BOOKSTORE"),
        OCI_ATTR_CONNECTION_CLASS, errhp);

    /* Purity need not be set, as default is OCI_ATTR_PURITY_SELF for OCISessionPool connections */

    /* You can get a SCOTT session released by Mid-tier 2 */
    OCISessionGet(envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
        NULL, NULL, OCI_SESSGET_SPOOL);

    /* Database calls using the svchp obtained above */
```
OCIStmtExecute(...)

/* This releases the pooled server on the database for reuse */
OCISessionRelease (svchp, errhp, NULL, 0, OCI_DEFAULT);
}

/* Mid-tier is done - exiting */
OCISessionPoolDestroy (spoolhp, errhp, OCI_DEFAULT);

Example 9–7 and Example 9–8 show two deployment examples, each based on code in Example 9–6, in which code is deployed in 10 middle-tier hosts servicing the BOOKSTORE application.

For the first deployment example, assume that the database used is Oracle Database 11g (or earlier) in dedicated server mode but with DRCP not enabled. The client side has 11g libraries. Example 9–7 shows the connect string to use for this deployment. In this case, the application obtains dedicated server connections from the database.

Example 9–7  Connect String to Use for a Deployment in Dedicated Server Mode with DRCP Not Enabled

BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oraclehost.company.com)(PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)))

For the second deployment example, assume that DRCP is enabled on the Oracle Database 11g database. Now all the middle-tier processes can benefit from the pooling capabilities offered by DRCP. The database resource requirement with DRCP is much less than what would be required with dedicated server mode. Example 9–8 shows how you change the connect string for this type of deployment.

Example 9–8  Connect String to Use for a Deployment with DRCP Enabled

BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oraclehost.company.com)(PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)(SERVER=POOLED)))

Compatibility and Migration

An OCI application linked with Oracle Database 11g client libraries works unaltered against:

- An Oracle Database 11g database with DRCP disabled
- A database server from a release earlier than Oracle Database 11g
- An Oracle Database 11g database server with DRCP enabled, when deployed with the DRCP connect string

Suitable clients benefit from enhanced scalability offered by DRCP if they are appropriately modified to use the OCI session pool APIs with the connection class and purity settings as previously described.

Restrictions on Using Database Resident Connection Pooling

The following actions cannot be performed or used with pooled servers:

- Shutting down the database
- Stopping the database resident connection pool
- Change the password for the connected user
Using shared database links to connect to a database resident connection pool that is on a different instance

Using Advanced Security Options (ASO) with TCPS

Using migratable sessions on the server side, directly by using the `OCI_MIGRATE` option or indirectly by using the `OCISessionPoolCreate()` call

Using initial client roles

Using Application context attributes such as `OCI_ATTR_APPCTX_NAME` and `OCI_ATTR_APPCTX_VALUE`

DDL statements that pertain to database users in the pool need to be performed carefully, as the pre-DDL sessions in the pool can still be given to clients post-DDL. For example, while dropping users, ensure that there are no sessions of that user in the pool and no connections to the Broker that were authenticated as that user.

Sessions with explicit roles enabled, that are released to the pool, can be later handed out to connections (of the same user) that need the default logon role. Avoid releasing sessions with explicit roles, and instead terminate them.

---

**Note:** You can use Oracle Advanced Security features such as encryption and strong authentication with DRCP.

---

**Using DRCP with Custom Pools**

DRCP is well integrated with OCI session pooling as described in "Database Resident Connection Pooling" on page 9-13. Oracle highly recommends using OCI session pool as it is already integrated with DRCP, FAN, and RLB.

However, if an application is built using its own custom connection pool (or if the application does not use any pooling at all, but has periods of time when the session is not used and the application does not depend on getting back the specific session for correctness), it can still integrate with DRCP. You can do this by enabling the `OCI_ATTR_SESSION_STATE` attribute as described in "Marking Sessions Explicitly as Stateful or Stateless" on page 9-21.

When an application flags a session as being `OCI_SESSION_STATELESS`, OCI benefits from this session annotation to return the session transparently to the DRCP pool (when DRCP is enabled). Similarly, when the application indicates the session as being `OCI_SESSION_STATEFUL`, OCI benefits from this changed session state annotation to transparently check out an appropriate session from the DRCP pool.

Applications should mark session state as promptly as possible to enable efficient utilization of the underlying database resources.

---

**Note:** Other DRCP attributes such as connection class and purity still must be specified as previously described in detail.

---

**Marking Sessions Explicitly as Stateful or Stateless**

An application typically requires a specific database session for the duration of a unit of work. The session is said to be STATEFUL for this duration. At the end of this unit of work, if the application does not depend on retaining the specific session for subsequent units of work, the session is said to be STATELESS.
As the application detects when a session transitions from being STATEFUL to STATELESS and vice versa, the application can explicitly inform OCI regarding these transitions by using *OCI_ATTR_SESSION_STATE*.

This indication by the application or caller can allow OCI and Oracle Database to take advantage of this information for transparently performing certain scalability optimizations. For example, the session could be given to someone else who needs it when the application is not working on it. Or, the session could be replaced by a different session when the application needs it again.

**See Also:** "Using DRCP with Custom Pools" on page 9-21

Here is an example of marking the state of sessions in a code fragment:

```c
wait_for_transaction_request();
do {

    ub1 state;

    /* mark database session as STATEFUL */
    state = OCI_SESSION_STATEFUL;
    checkerr(errhp, OCIAttrSet(usrhp, OCI_HTYPE_SESSION,
                        &state, 0, OCI_ATTR_SESSION_STATE, errhp));
    /* do database work consisting of one or more related calls to the database */
    ...

    /* done with database work, mark session as stateless */
    state = OCI_SESSION_STATELESS;
    checkerr(errhp, OCIAttrSet(usrhp, OCI_HTYPE_SESSION,
                        &state, 0, OCI_ATTR_SESSION_STATE, errhp));

    wait_for_transaction_request();
} while(not _done);
```

If a session is obtained from outside an OCI session pool, the session starts as *OCI_SESSION_STATEFUL* and remains *OCI_SESSION_STATEFUL* throughout the life of the session unless the application explicitly changes it to *OCI_SESSION_STATELESS*.

If a session is obtained from an OCI session pool, the session is by default marked as *OCI_SESSION_STATEFUL* when the first call is initiated on that session after getting it from the pool. The session is also by default marked as being *OCI_SESSION_STATELESS* when it is released to the pool. Hence, there is no need to set this attribute explicitly with an OCI session pool. OCI session pooling does this transparently. Use this attribute only if you are not using OCI session pooling.

**See Also:** "OCI_ATTR_SESSION_STATE" on page A-21

**DRCP with Real Application Clusters**

When the Database Resident Connection Pool is configured in a database in a Real Application Clusters environment, the pool configuration is applied to each of the database instances. Starting or stopping of the pool on one instance will start or stop the pool on all the instances.
DRCP with Data Guard

The Database Resident Connection Pool has certain conditions to operate in a Data Guard environment.

- Starting the pool: The pool can be started on a physical standby database only if the pool has been already started on the primary database. If the pool is down on the primary, it cannot be started on the standby database.
- Stopping the pool: The pool cannot be stopped on a physical standby database if it is up and running on the primary database. It can only be stopped if on the primary database, the pool is not running.
- Pool parameters cannot be configured, restored to defaults, or altered on a physical standby database.
- When role reversal takes place, that is, the Primary goes down and the Secondary database takes up the role of the Primary database, the limitations mentioned previously for the physical standby database no longer holds. Since the Standby database has now become the Primary, all pool operations are allowed.
- On a logical standby database, all pool operations are allowed.

When to Use Connection Pooling, Session Pooling, or Neither

This section includes the following topics:

- Functions for Session Creation
- Choosing Between Different Types of OCI Sessions

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.

**Functions for Session Creation**

OCI offers the following functions for session creation:

- **OCILogon()**
  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.

  **See Also:**
  - "OCILogon()" on page 16-22
  - "Application Initialization, Connection, and Session Creation" on page 2-14

- **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.

  **See Also:** "OCILogon2()" on page 16-24

- **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.
When to Use Connection Pooling, Session Pooling, or Neither

OCI Programming Advanced Topics  9-25

See Also:  "OCISessionBegin()" on page 16-30

- 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:  "OCISessionGet()" on page 16-34

Choosing Between Different Types of OCI Sessions

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.
Statement Caching in OCI

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

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()

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:

```c
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: "Middle-Tier Applications in OCI" on page 8-10

- Migratable Sessions

With transaction handles being migratable, there should be no need for applications to use migratable sessions, instead use OCI connection pooling.

See Also: "OCI Session Management" on page 8-9
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.

---

Rules for Statement Caching in OCI

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:**
- "Statement Functions" on page 17-2
- "Service Context Handle Attributes" on page A-8
- "Session Pool Handle Attributes" on page A-24

**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 9–9 shows the pseudocode.

**Example 9–9  Optimizing Bind and Define Operations on Statements in the Cache**

Get the statement using OCIStrmPrepare2(...)

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.

OCIStrmRelease() that statement

Next OCIStrmPrepare2()

OCIAttrGet() opaque application context from statement handle

Execute/Fetch using the statement and process the data in the fetch buffers.

OCIStrmRelease()

```
...
```

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" on page A-33
- "OCI_ATTR_STMTCACHE_CBK" on page A-10

**OCI Statement Caching Code Example**

See cdemostc.c in directory demo for a working example of statement caching.

**User-Defined Callback Functions in OCI**

This section includes the following topics:
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.

### 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.

- 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.

See Also:
- "OCIUserCallbackGet()" on page 17-177
- "OCIUserCallbackRegister()" on page 17-179
A user callback can return `OCI_INVALID_HANDLE` when either an invalid handle or an invalid context is passed to it.

---

**Note:** If any callback returns anything other than `OCI_CONTINUE`, then that return code is passed to the subsequent callbacks. If a replacement or exit callback returns a return code other than `OCI_CONTINUE`, then the final (not `OCI_CONTINUE`) return code is returned from the OCI call.

---

**OCIUserCallbackRegister**

A user callback is registered using the `OCIUserCallbackRegister()` call.

See Also:  "OCIUserCallbackRegister()" on page 17-179

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_dynckb_fn` entry callback function and its context `dynamic_context`, are registered against the environment handle `hndlp` for the `OCIStmtPrepare()` call by calling the `OCIUserCallbackRegister()` function with the following parameters.

```c
OCIUserCallbackRegister( hndlp, 
                      OCI_HTYPE_ENV, 
                      errh, 
                      stmtprep_entry_dynckb_fn, 
                      dynamic_context, 
                      OCI_FNCODE_STMTPREPARE, 
                      OCI_UCBTYPE_ENTRY 
                      (OCIUcb*) NULL);
```

**User Callback Function**

The user callback function must use the following syntax:

```c
typedef sword (*OCIUserCallback)
   (void *ctxp,      /* context for the user callback*/
    void *hndlp,     /* handle for the callback, env handle for now */
    ub4 type,         /* type of handlep, 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 *ernnop,       /* 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, `errno`, 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 `*ernnop` is always passed in as 0 for the first entry callback. Note that `*ernnop` refers to the content of `ernnop` because `ernnop` 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 `*ernnop` 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.

See Also: Appendix B, “OCI Demonstration Programs”

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.

User Callback Control Flow

Example 9–10 shows pseudocode that describes the overall processing of a typical OCI call.

Example 9–10  Pseudocode That Describes the Overall Processing of a Typical OCI Call

OCIYyzCall()
{
    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 */

erro = error number from error handle or env handle;

executeExitCallback:
  for all EXIT callbacks do
  {
    exitRetCode = (*exitCallback)(..., retCode, &erro, ...);
    if (exitRetCode != OCI_CONTINUE)
    {
      set erro in error handle or environment handle;
      retCode = exitRetCode;
    }
  }
  release mutexes;
  return retCode
}

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.

Errors from Entry Callbacks

If an entry callback wants to return an error to the caller of the OCI call, then it must register a replacement or exit callback. This is because if the OCI code is executed, then the error code from the entry callback is ignored. Therefore, the entry callback must pass the error to the replacement or exit callback through its own context.

Dynamic Callback Registrations

Because user callbacks are expected to be used for monitoring OCI behavior or to access other data sources, it is desirable that the registration of the callbacks be done transparently and 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.
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.

---

Package Format

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 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:

```c
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 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:

```c
typedef sword (*OCIEnvCallbackType)(OCIEnv *env, ub4 mode,
```
When an environment handle is created, then this callback function is called at the
very end. The env parameter is the newly created environment handle.

The mode, xtramem_sz, and usrmemp are the parameters passed to the OCIEnvCreate()
call. The last parameter, ucbDesc, is a descriptor that is passed to the package. The
package uses this descriptor to register the user callbacks as described later.

A sample ociucb.c file is provided in the demo directory. The makefile ociucb.mk is
also provided (on Solaris) in the demo directory to create the package. Please note that
this may be different on other operating systems. The demo directory also contains full
user callback demo programs (cdemouch.c, cdemouchl.c) illustrating this.

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()" on page 17-177

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.

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
  OCICommit(), then the callback for OCICommit() is disabled during the
  execution of the callback for OCICommit().

- 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()
User-Defined Callback Functions in OCI

- OCIEnvCreate()
- OCIInitialize() (Deprecated)
- OCIEnvCreate()

**Example of OCI Callbacks**

Suppose that there are five packages each registering entry, replacement, and exit callbacks for the OCIStmtPrepare() call. That is, the ORA_OCI_UCBPKG variable is set as shown in Example 9–11.

**Example 9–11 Environment Variable Setting for the ORA_OCI_UCBPKG Variable**

```c
setenv ORA_OCI_UCBPKG "pkg1;pkg2;pkg3;pkg4;pkg5"
```

In each package pkgN (where N can be 1 through 5), the pkgNInit() and PkgNEnvCallback() functions are specified, as shown in Example 9–12.

**Example 9–12 Specifying the pkgNInit() and PkgNEnvCallback() Functions**

```c
def pkgNInit(void *metaCtx, void *libCtx, ub4 argfmt, sword argc, void **argv)
{
    return OCISharedLibInit(metaCtx, libCtx, argfmt, argc, argv, pkgNEnvCallback);
}
```

Example 9–13 shows how the pkgNEnvCallback() function registers the entry, replacement, and exit callbacks.

**Example 9–13 Using pkgNEnvCallback() to Register Entry, Replacement, and Exit Callbacks**

```c
def 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;
}
```

Finally, Example 9–14 shows how in the source code for the application, user callbacks can be registered with the NULL ucbDesc.

**Example 9–14 Registering User Callbacks with NULL ucbDesc**

```c
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);
```
OCI_USER_CALLBACK_REGISTER\(\text{env, OCI\_HTYPE\_ENV, errh, static\_exit\_callback\_fn, pkgNctx, OCI\_FNCODE\_STMTPREPARE, OCI\_UCBTYPE\_EXIT, (OCI\_Ucb *)NULL}\);

Example 9–15 shows that when the \texttt{OCIStmtPrepare()} call is executed, the callbacks are called in the following order.

**Example 9–15 Using the OCIStmtPrepare() Call to Call the Callbacks in Order**

\[
\begin{align*}
\text{static\_entry\_callback\_fn}() \\
\text{pkg1\_entry\_callback\_fn}() \\
\text{pkg2\_entry\_callback\_fn}() \\
\text{pkg3\_entry\_callback\_fn}() \\
\text{pkg4\_entry\_callback\_fn}() \\
\text{pkg5\_entry\_callback\_fn}() \\
\text{static\_replace\_callback\_fn}() \\
\text{pkg1\_replace\_callback\_fn}() \\
\text{pkg2\_replace\_callback\_fn}() \\
\text{pkg3\_replace\_callback\_fn}() \\
\text{pkg4\_replace\_callback\_fn}() \\
\text{pkg5\_replace\_callback\_fn}() \\
\text{OCI code for OCIStmtPrepare call} \\
\text{pkg5\_exit\_callback\_fn}() \\
\text{pkg4\_exit\_callback\_fn}() \\
\text{pkg3\_exit\_callback\_fn}() \\
\text{pkg2\_exit\_callback\_fn}() \\
\text{pkg1\_exit\_callback\_fn}() \\
\text{static\_exit\_callback\_fn}()
\end{align*}
\]

**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 \texttt{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 \texttt{pkg4\_replace\_callback\_fn()} returned \texttt{OCI\_SUCCESS}, then \texttt{pkg4\_replace\_callback\_fn()}, \texttt{pkg5\_replace\_callback\_fn()}, and the OCI processing for the \texttt{OCIStmtPrepare()} call are bypassed. Instead, \texttt{pkg5\_exit\_callback\_fn()} is executed next.

**OCI Callbacks from External Procedures**

There are several OCI functions that you can use as callbacks from external procedures. These functions are listed in Chapter 20. 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, see *Oracle Database Advanced Application Developer’s Guide*.

**Transparent Application Failover in OCI**

This section includes the following topics:

- Configuring Transparent Application Failover
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. Prior to Oracle Database 10g Release 2 (10.2), TAF with the SELECT failover option would be engaged only on the statement that was in use at the time of a failure. For example, if there were 10 statement handles in use by the application, and statement 7 was the failure-time statement (the statement in use when the failure happened), statements 1 through 6 and 8 through 10 would have to be reexecuted after statement 7 was failed over using TAF.

Starting with Oracle Database 10g Release 2 (10.2), this has been improved. Now 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:** TAF is not supported for remote database links or for DML statements.

**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.

**See Also:** Oracle Database Net Services Reference for more information about client-side configuration of TAF (Connect Data Section)

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 PL/SQL Packages and Types Reference for more information about the server-side configuration of TAF (DBMS_SERVICE)

**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: "Handling OCI_FO_ERROR" on page 9-42 for more information about this scenario

**Failover Callback Structure and Parameters**

The basic structure of a user-defined application failover callback function is as follows:

```c
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 9-40 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.
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).

### Failover Callback Registration

For the failover callback to be used, it must be registered on the server context handle. This registration is done by creating a callback definition structure and setting the **OCI_ATTR_FOCBK** attribute of the server handle to this structure.

The callback definition structure must be of type **OCIFocbkStruct**. It has two fields:
- **callback_function**, which contains the address of the function to call, and
- **fo_ctx**, which contains the address of the client context.

An example of callback registration is included as part of Example 9–17.

### Failover Callback Example

This section shows an example of a simple user-defined callback function definition (see Example 9–16), failover callback registration (see Example 9–17), and failover callback unregistration (see Example 9–18).

#### Example 9–16  User-Defined Failover Callback Function Definition

```c
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:
        { /* Failing Over ... Please stand by */
            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
",
                (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 9–17  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 9–18  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);
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 9–2.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>Database fails (failure lasts until T5).</td>
</tr>
<tr>
<td>T1</td>
<td>Failover is triggered by user activity.</td>
</tr>
<tr>
<td>T2</td>
<td>User attempts to reconnect; attempt fails.</td>
</tr>
<tr>
<td>T3</td>
<td>Failover callback is invoked with OCI_FO_ERROR.</td>
</tr>
<tr>
<td>T4</td>
<td>Failover callback enters a predetermined sleep period.</td>
</tr>
<tr>
<td>T5</td>
<td>Database comes back up again.</td>
</tr>
<tr>
<td>T6</td>
<td>Failover callback triggers a new failover attempt; it is successful.</td>
</tr>
<tr>
<td>T7</td>
<td>User successfully reconnects.</td>
</tr>
</tbody>
</table>

The callback function triggers the new failover attempt by returning a value of OCI_FO_RETRY from the function.

Example 9–19 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:

```c
Example 9–19 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;
  OCIBufferAlloc(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
",
               (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;

HA Event Notification

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

Determining Transparent Application Failover (TAF) Capabilities

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.

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.

**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" on page A-77

**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.

**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.

**Event Callback**

The event callback, of type `OCIEventCallback`, has the following signature:

```c
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.

```c
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`:

```c
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 \texttt{OCI\_ATTR\_HA\_SRVFIRST}, this function retrieves the first server handle in the list of server handles affected. When called with attribute \texttt{OCI\_ATTR\_HA\_SRVNEXT}, this function retrieves the next server handle in the list. This function returns \texttt{OCI\_NO\_DATA} and \texttt{srvhp} is a \texttt{NULL} pointer, when there are no more server handles to return.

\texttt{srvhp} is an output pointer to a server handle whose connection has been closed because of an HA event. \texttt{errhp} is an error handle to populate. The application returns an \texttt{OCI\_NO\_DATA} error when there are no more affected server handles to retrieve.

When retrieving the list of server handles that have been affected by an HA event, be aware that the connection has already been closed and many server handle attributes are no longer valid. Instead, use the user memory segment of the server handle to store any per-connection attributes required by the event notification callback. This memory remains valid until the server handle is freed.

**Custom Pooling: Tagged Server Handles**

The following features apply to custom pools:

- 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 \texttt{OCIHandleAlloc()} to request that the server handle be allocated with a user memory segment. A pointer to the "user memory" segment is returned by \texttt{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 \texttt{OCI\_ATTR\_USER\_MEMORY} is used to retrieve a pointer to a handle’s user memory segment. \texttt{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 \texttt{NULL} pointer is returned for those handles not allocated with extra memory. This attribute is read-only and is of data type \texttt{void*}.

\textbf{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 \texttt{OCIHandleFree()} call on the server handle.

Example 9–20 shows an example of event notification.

**Example 9–20  Event Notification**

```c
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,
```
/* 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;
    s_int 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)OCI_ATTR_USER_MEMORY, errhp);
        /* Remove the server handle from the parent connection object */
        retcode = OCIAttrGet(eventhp, OCI_HTYPE_SERVER, &srvhp, (ub4 *)0,
                              OCI_ATTR_HA_SRVNEXT, errhp);
        ...
    }
    ...
}

Determining Transparent Application Failover (TAF) Capabilities
You can have the application adjust its behavior if a connection is or is not
TAF-enabled. Use OCIAttrGet() as follows to 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.

OCI and Streams Advanced Queuing
This section includes the following topics:
- OCI Streams Advanced Queuing Functions
- OCI Streams Advanced Queuing Descriptors
- Streams Advanced Queuing in OCI Versus PL/SQL
- Buffered Messaging

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

---

**Note:** To use Streams Advanced Queuing, you must be using the Enterprise Edition of Oracle Database.

---

**See Also:**

- *Oracle Streams Advanced Queuing User’s Guide*
- *Oracle XML Developer’s Kit Programmer’s Guide*
- The description of "OCIAQEnq()" on page 17-93 for example code demonstrating the use of OCI with AQ

### OCI Streams Advanced Queuing Functions

The OCI library includes several functions related to Streams 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:** "Streams Advanced Queuing and Publish-Subscribe Functions" on page 17-88

### OCI Streams Advanced Queuing Descriptors

The following descriptors are used by OCI Streams AQ operations:

- OCIAQEnqOptions
- OCIAQDeqOptions
- OCIAQMsgProperties
- OCIAQAgent

You can allocate these descriptors with the service handle using the standard `OCIDescriptorAlloc()` call. The following code shows examples of this:

```c
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.
Streams Advanced Queuing in OCI Versus PL/SQL

The following tables compare functions, parameters, and options for OCI Streams AQ functions and descriptors, and PL/SQL AQ functions in the DBMS_AQ package. Table 9–3 compares AQ functions.

Table 9–3 AQ Functions

<table>
<thead>
<tr>
<th>PL/SQL Function</th>
<th>OCI Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_AQ.ENQUEUE</td>
<td>OCIAQEnq()</td>
</tr>
<tr>
<td>DBMS_AQ.DEQUEUE</td>
<td>OCIAQDeq()</td>
</tr>
<tr>
<td>DBMS_AQ.LISTEN</td>
<td>OCIAQListen(), OCIAQListen2()</td>
</tr>
<tr>
<td>DBMS_AQ.ENQUEUE_ARRAY</td>
<td>OCIAQEnqArray()</td>
</tr>
<tr>
<td>DBMS_AQ.DEQUEUE_ARRAY</td>
<td>OCIAQDeqArray()</td>
</tr>
</tbody>
</table>

Table 9–4 compares the parameters for the enqueue functions.

Table 9–4 Enqueue Parameters

<table>
<thead>
<tr>
<th>DBMS_AQ.ENQUEUE Parameter</th>
<th>OCIAQEnq() Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue_name</td>
<td>queue_name</td>
</tr>
<tr>
<td>enqueue_options</td>
<td>enqueue_options</td>
</tr>
<tr>
<td>message_properties</td>
<td>message_properties</td>
</tr>
<tr>
<td>payload</td>
<td>payload</td>
</tr>
<tr>
<td>msgid</td>
<td>msgid</td>
</tr>
</tbody>
</table>

Note: OCIAQEnq() requires the following additional parameters: svch, errh, payload_tdo, payload_ind, and flags.

Table 9–5 compares the parameters for the dequeue functions.

Table 9–5 Dequeue Parameters

<table>
<thead>
<tr>
<th>DBMS_AQ.DEQUEUE Parameter</th>
<th>OCIAQDeq() Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue_name</td>
<td>queue_name</td>
</tr>
<tr>
<td>dequeue_options</td>
<td>dequeue_options</td>
</tr>
<tr>
<td>message_properties</td>
<td>message_properties</td>
</tr>
<tr>
<td>payload</td>
<td>payload</td>
</tr>
<tr>
<td>msgid</td>
<td>msgid</td>
</tr>
</tbody>
</table>

Note: OCIAQDeq() requires the following additional parameters: svch, errh, dequeue_options, message_properties, payload_tdo, payload, payload_ind, and flags.

Table 9–6 compares parameters for the listen functions.
### Table 9–6  Listen Parameters

<table>
<thead>
<tr>
<th>DBMS_AQ.LISTEN Parameter</th>
<th>OCIAQListen2() Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent_list</td>
<td>agent_list</td>
</tr>
<tr>
<td>wait</td>
<td>wait</td>
</tr>
<tr>
<td>agent</td>
<td>agent</td>
</tr>
<tr>
<td>listen_delivery_mode</td>
<td>lopts</td>
</tr>
</tbody>
</table>

[Note: OCIAQListen2() requires the following additional parameters: svchp, errhp, agent_list, num_agents, agent, lmops, and flags.]

### Table 9–7  Array Enqueue Parameters

<table>
<thead>
<tr>
<th>DBMS_AQ.ENQUEUE_ARRAY Parameter</th>
<th>OCIAQEnqArray() Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue_name</td>
<td>queue_name</td>
</tr>
<tr>
<td>enqueue_options</td>
<td>enqopt</td>
</tr>
<tr>
<td>array_size</td>
<td>iters</td>
</tr>
<tr>
<td>message_properties_array</td>
<td>msgprop</td>
</tr>
<tr>
<td>payload_array</td>
<td>payload</td>
</tr>
<tr>
<td>msgid_array</td>
<td>msgid</td>
</tr>
</tbody>
</table>

[Note: OCIAQEnqArray() requires the following additional parameters: svch, errh, payload_tdo, payload_ind, ctxp, enqcbfp, and flags.]

### Table 9–8  Array Dequeue Parameters

<table>
<thead>
<tr>
<th>DBMS_AQ.DEQUEUE_ARRAY Parameter</th>
<th>OCIAQDeqArray() Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue_name</td>
<td>queue_name</td>
</tr>
<tr>
<td>dequeue_options</td>
<td>deqopt</td>
</tr>
<tr>
<td>array_size</td>
<td>iters</td>
</tr>
<tr>
<td>message_properties_array</td>
<td>msgprop</td>
</tr>
<tr>
<td>payload_array</td>
<td>payload</td>
</tr>
<tr>
<td>msgid_array</td>
<td>msgid</td>
</tr>
</tbody>
</table>

[Note: OCIAQDeqArray() requires the following additional parameters: svch, errh, msgprop, payload_tdo, payload_ind, ctxp, deqcbfp, and flags.]

### Table 9–9  Agent Parameters

<table>
<thead>
<tr>
<th>PL/SQL Agent Parameter</th>
<th>OCIAQAgent Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>OCI_ATTR_AGENT_NAME</td>
</tr>
</tbody>
</table>
Table 9–10 compares parameters for the message properties.

<table>
<thead>
<tr>
<th>PL/SQL Message Property</th>
<th>OCIAQMsgProperties Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>OCI_ATTR_PRIORITY</td>
</tr>
<tr>
<td>delay</td>
<td>OCI_ATTR_DELAY</td>
</tr>
<tr>
<td>expiration</td>
<td>OCI_ATTR_EXPIRATION</td>
</tr>
<tr>
<td>correlation</td>
<td>OCI_ATTR_CORRELATION</td>
</tr>
<tr>
<td>attempts</td>
<td>OCI_ATTR_ATTEMPTS</td>
</tr>
<tr>
<td>recipient_list</td>
<td>OCI_ATTR_RECIPIENT_LIST</td>
</tr>
<tr>
<td>exception_queue</td>
<td>OCI_ATTR_EXCEPTION_QUEUE</td>
</tr>
<tr>
<td>enqueue_time</td>
<td>OCI_ATTR_ENQ_TIME</td>
</tr>
<tr>
<td>state</td>
<td>OCI_ATTR_MSG_STATE</td>
</tr>
<tr>
<td>sender_id</td>
<td>OCI_ATTR_SENDER_ID</td>
</tr>
<tr>
<td>transaction_group</td>
<td>OCI_ATTR_TRANSACTION_NO</td>
</tr>
<tr>
<td>original_msgid</td>
<td>OCI_ATTR_ORIGINAL_MSGID</td>
</tr>
<tr>
<td>delivery_mode</td>
<td>OCI_ATTR_MSG_DELIVERY_MODE</td>
</tr>
</tbody>
</table>

Table 9–11 compares enqueue option attributes.

<table>
<thead>
<tr>
<th>PL/SQL Enqueue Option</th>
<th>OCIAQEnqOptions Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>visibility</td>
<td>OCI_ATTR_VISIBILITY</td>
</tr>
<tr>
<td>relative_msgid</td>
<td>OCI_ATTR_RELATIVE_MSGID</td>
</tr>
<tr>
<td>sequence_deviation</td>
<td>OCI_ATTR_SEQUENCE_DEVIATION</td>
</tr>
<tr>
<td></td>
<td>(deprecated)</td>
</tr>
<tr>
<td>transformation</td>
<td>OCI_ATTR_TRANSFORMATION</td>
</tr>
<tr>
<td>delivery_mode</td>
<td>OCI_ATTR_MSG_DELIVERY_MODE</td>
</tr>
</tbody>
</table>

Table 9–12 compares dequeue option attributes.

<table>
<thead>
<tr>
<th>PL/SQL Dequeue Option</th>
<th>OCIAQDeqOptions Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>consumer_name</td>
<td>OCI_ATTR_CONSUMER_NAME</td>
</tr>
<tr>
<td>dequeue_mode</td>
<td>OCI_ATTR_DEQ_MODE</td>
</tr>
<tr>
<td>navigation</td>
<td>OCI_ATTR_NAVIGATION</td>
</tr>
<tr>
<td>visibility</td>
<td>OCI_ATTR_VISIBILITY</td>
</tr>
</tbody>
</table>
Buffered Messaging

Buffered messaging is a nonpersistent messaging capability within Streams AQ that was first available in Oracle Database 10g 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()" on page 17-93
- "OCIAQDeq()" on page 17-89
- "OCIAQListen2()" on page 17-97

Example 9–21 shows an example of enqueue buffered messaging.

Example 9–21 Enqueue Buffered Messaging

```c
... 
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 9–22 shows an example of dequeue buffered messaging.

**Example 9–22  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);
...

**Note:** Array operations are not supported for buffered messaging.
Applications can use the OCIALQEnqArray() and OCIALQDeqArray() functions with the array size set to 1.
Publish-Subscribe Notification in OCI

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

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 9–2 illustrates the process.

**Figure 9–2  Publish-Subscribe Model**

An OCI application can:

- Register interest in notifications in the AQ namespace and be notified when an enqueue occurs
- Register interest in subscriptions to database events and receive notifications when the events are triggered
Manage registrations, such as disabling registrations temporarily or dropping the registrations entirely

Post or send notifications to registered clients

In all the preceding scenarios the notification can be received directly by the OCI application, or the notification can be sent to a 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.

See Also:
- "OCI and Streams Advanced Queuing" on page 9-47 for information about Streams Advanced Queuing
- Oracle Streams Advanced Queuing User’s Guide for information about creating queues and about Streams AQ, including concepts, features, and examples
- The chapter about CREATE TRIGGER in the Oracle Database SQL Language Reference for information about creating triggers

Publish-Subscribe Registration Functions in OCI

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. See "Publish-Subscribe Register Directly to the Database" on page 9-55.

- 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. See "Open Registration for Publish-Subscribe" on page 9-58.

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 higher.

See Also:
- "Streams Advanced Queuing and Publish-Subscribe Functions” on page 17-88
- "Publish-Subscribe Direct Registration Example" on page 9-64 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 `OCIAtrrSet()` 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.


   - `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:
     ```c
     #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:

```c
#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" on page A-51 and "Creating the OCI Environment" on page 2-13 for setting up the environment with `mode = OCI_EVENTS | OCI_OBJECT`. **OCI_OBJECT** is required for grouping notifications.

4. Set the values of QOS, timeout interval, namespace, and port (see Example 9–23).

**See Also:** "Setting QOS, Timeout Interval, Namespace, Client Address, and Port Number" on page 9-60

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" on page 9-61

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" on page 9-64

7. Call **OCISubscriptionRegister()** to register with the subscriptions. This call can register interest in several subscriptions at the same time.

Example 9–23 shows an example of setting QOS levels.

**Example 9–23  Setting QOS Levels, the Notification Grouping Class, Value, and Type, and the Namespace Specific Context**

```c
/* 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,
(void) &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,
(void) &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,
(void) &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,
(void) NULL, (ub4) 0,
(ub4) OCI_ATTR_SUBSCR_NAMESPACE_CTX, errhp);

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 Advanced Security Administrator’s Guide, sections about managing enterprise user security

- The compatibility of the database must be 9.0 or higher.

- LDAP_REGISTRATION_ENABLED must be set to TRUE. This can be done this way:
  
  ALTER SYSTEM SET LDAP_REGISTRATION_ENABLED=TRUE

  The default is FALSE.

- LDAP_REG_SYNC_INTERVAL must be set to the time interval (in seconds) to refresh registrations from LDAP:
  
  ALTER SYSTEM SET LDAP_REG_SYNC_INTERVAL = time_interval

  The default is 0, which means do not refresh.

- To force a database refresh of LDAP registration information immediately:
  
  ALTER SYSTEM REFRESH LDAP_REGISTRATION

The steps 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`.

Using OCI to Open Register with LDAP

1. Call `OCIEnvCreate()` or `OCIEnvNlsCreate()` with mode set to `OCI_EVENTS | OCI_USE_LDAP`.

2. Call `OCIArrAttrSet()` 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_NALL_LOC`: for SSL authentication, the location of the client wallet
   - `OCI_ATTR_LDAP_AUTH`: the authentication method code

   See Also: "Environment Handle Attributes" on page A-2 for a complete list of authentication modes

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 `OCIArrAttrSet()` 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. `OCIArrAttrSet()` can be called multiple times for this attribute so that more than one database server is included in the registration.

6. Call `OCIArrAttrSet()` 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_RECEPTRRES`: presentation to receive notification with
**Publish-Subscribe Notification in OCI**

- **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" on page 9-60.

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`.

---

**Setting QOS, Timeout Interval, Namespace, Client Address, and Port Number**

You can set QOSFLAGS to the following QOS levels using `OCIAttrSet()`:

- **OCI_SUBSCR_QOS_RELIABLE** - Reliable notification persists across instance and database restarts. Reliability is of the server only and is only for persistent queues or buffered messages. This option describes the persistence of the notifications. Registrations are persistent by default.
- **OCI_SUBSCR_QOS_PURGE_ON_NTFN** - Once notification is received, purge registration on first notification. (Subscription is unregistered.)

```c
/* Set QOS levels */
ub4 qosflags = OCI_SUBSCR_QOS_RELIABLE | OCI_SUBSCR_QOS_PURGE_ON_NTFN;

/* Set flags in subscription handle */
(void)OCIAttrSet((void *)subchrhp, (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 *)subchrhp, (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.

```c
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.

```c
(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" on page A-53

### OCI Functions Used to Manage Publish-Subscribe Notification

Table 9–13 lists the functions that are used to manage publish-subscribe notification.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCISubscriptionDisable()</td>
<td>Disables a subscription</td>
</tr>
<tr>
<td>OCISubscriptionEnable()</td>
<td>Enables a subscription</td>
</tr>
<tr>
<td>OCISubscriptionPost()</td>
<td>Posts a subscription</td>
</tr>
<tr>
<td>OCISubscriptionRegister()</td>
<td>Registers a subscription</td>
</tr>
<tr>
<td>OCISubscriptionUnRegister()</td>
<td>Unregisters a subscription</td>
</tr>
</tbody>
</table>

### 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" on page A-51

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 Streams Advanced Queuing" on page 9-47
- "Notification Descriptor Attributes" on page A-60

**iMode (IN)**
Call-specific mode. The only valid value is OCI_DEFAULT. This value executes the default call.

Example 9–24 shows how to use AQ grouping notification attributes in a notification callback.

**Example 9–24  Using AQ Grouping Notification Attributes in an OCI Notification Callback**

```c
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                 nummsgid = 0;
  void               *elemind = (void *)0;
  boolean             exist;
  ub2                 flags;
  oratext            *hexit = (oratext *)&quot;0123456789ABCDEF&quot;;
  ub4                 i, j;

  /* get subscription name */
  OCIAttrGet(subscrhp, OCI_HTYPE_SUBSCRIPTION, (void *)&#034;subname&#034;, &size,
  OCI_ATTR_SUBSCR_NAME, ctxptr-&gt;errhp);

  /* print subscription name */
  printf(&quot;Got notification for %.*s\n", size, subname);
  fflush((FILE *)__stdout);
}
```

See Also:
- "OCI and Streams Advanced Queuing" on page 9-47
- "Notification Descriptor Attributes" on page A-60
/* 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++)
{
  /* raw size */
  ub4  rawSize;
  /* raw pointer */
  ub1 *rawPtr;
  /* 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;
}

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" on page A-51
- "Oracle Streams AQ PL/SQL Callback" in Oracle Database
  PL/SQL Packages and Types Reference for the PL/SQL procedure specification

Publish-Subscribe Direct Registration Example

Example 9–25 shows how system events, client notification, and Advanced Queuing work together to implement publish subscription notification.

The PL/SQL code in Example 9–25 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 9–25.

Example 9–25  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;
After the subscriptions are created, the client must register for notification using callback functions. Example 9–26 shows sample code that performs the necessary steps for registration. The initial steps of allocating and initializing session handles are omitted here for clarity.

**Example 9–26  Registering for Notification Using Callback Functions**

```c
... 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_RECVPROTO, 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("Beginning 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);

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.

Publish-Subscribe LDAP Registration Example

Example 9–27 shows a code fragment that illustrates how to do LDAP registration. Please read all the program comments.

Example 9–27    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);
You can use OCI to access Oracle TimesTen In-Memory Database and Oracle TimesTen Application-Tier Database Cache. See Oracle TimesTen In-Memory Database C Developer’s Guide, for information about Times Ten support for Oracle Call Interface.

This chapter contains these topics:

- Continuous Query Notification
- Database Startup and Shutdown
- Implicit Fetching of ROWIDs
- Client Result Cache
- Fault Diagnosability in OCI
- Client and Server Operating with Different Versions of Time Zone Files

Continuous Query Notification

Continuous Query Notification 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.

See Also: Oracle Database Advanced Application Developer’s Guide, "Using Continuous Query Notification" for a complete discussion of the concepts of this feature

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.
Continuous Query Notification

- Names of the modified objects or changed rows.
- Operation type (INSERT, UPDATE, DELETE, ALTER TABLE, DROP TABLE).
- ROWIDs 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: "Publish-Subscribe Notification in OCI" on page 9-54

Using Query Result Set Notifications

To record QOS (quality of service flags) specific to continuous query (CQ) notifications, set the attribute OCI_ATTR_SUBSCR_CQ_QOSFLAGS on the subscription handle OCI_HTYPE_SUBSCR. To request that the registration is at query granularity, as opposed to object granularity, set the OCI_SUBSCR_CQ_QOS_QUERY flag bit on the attribute OCI_ATTR_SUBSCR_CQ_QOSFLAGS.

The pseudocolumn CQ_NOTIFICATION_QUERY_ID can be optionally specified to retrieve the query ID of a registered query. Note that this does not automatically convert the granularity to query level. The value of the pseudocolumn on return is set to the unique query ID assigned to the query. The query ID pseudocolumn can be omitted for OCI-based registrations, in which case the query ID is returned as a READ attribute of the statement handle. (This attribute is called OCI_ATTR_CQ_QUERYID).

During notifications, the client-specified callback is invoked and the top-level notification descriptor is passed as an argument.

Information about the query IDs of the changed queries is conveyed through a special descriptor type called OCI_DTYPE_CQDES. A collection (OCIColl) of query descriptors is embedded inside the top-level notification descriptor. Each descriptor is of type OCI_DTYPE_CQDES. The query descriptor has the following attributes:

- OCI_ATTR_CQDES_OPERATION - can be one of OCI_EVENT_QUERYCHANGE or OCI_EVENT_DEREG.
- OCI_ATTR_CQDES_QUERYID - query ID of the changed query.
- OCI_ATTR_CQDES_TABLE_CHANGES - array of table descriptors describing DML operations on tables that led to the query result set change. Each table descriptor is of the type OCI_DTYPE_TABLE_CHDES.

See Also: "OCI_DTYPE_CQDES" on page 10-6

Registering for Continuous Query Notification

The calling session must have the CHANGE_NOTIFICATION system privilege and SELECT privileges on all objects that it attempts to register. A registration is a persistent entity that is recorded in the database, and is visible to all instances of Oracle RAC. If the registration was at query granularity, transactions that cause the query result set to change and commit in any instance of Oracle RAC generate notification.

If the registration was at object granularity, transactions that modify registered objects in any instance of Oracle RAC generate notification.

Queries involving materialized views or nonmaterialized views are not supported.

The registration interface employs a callback to respond to changes in underlying objects of a query and uses a namespace extension (DBCHANGE) to AQ.
The steps in writing the registration are as follows:

1. Create the environment in OCI_EVENTS and OCI_OBJECT mode.
2. Set the subscription handle attribute OCI_ATTR_SUBSCR_NAMESPACE to namespace OCI_SUBSCR_NAMESPACE_DBCHANGE.
3. Set the subscription handle attribute OCI_ATTR_SUBSCR_CALLBACK to store the OCI callback associated with the query handle. The callback has the following prototype:

   ```c
   void notification_callback (void *ctx, OCISubscription *subscrhp,
                            void *payload, ub4 paylen, void *desc, ub4 mode);
   ```

   The parameters are described in "Notification Callback in OCI" on page 9-61.
5. Set the OCI_ATTR_SUBSCR_TIMEOUT attribute to specify a ub4 timeout interval in seconds. If it is not set, there is no timeout.
6. Set the OCI_ATTR_SUBSCR_QOSFLAGS attribute, the QOS (quality of service) levels, with the following values:
   - The OCI_SUBSCR_QOS_PURGE_ON_NTFN flag allows the registration to be purged on the first notification.
   - The OCI_SUBSCR_QOS_RELIABLE flag allows notifications to be persistent. You can use surviving instances of Oracle RAC to send and retrieve continuous query 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.
7. Call OCISubscriptionRegister() to create a new registration in the DBCHANGE namespace.
8. Associate multiple query statements with the subscription handle by setting the attribute OCI_ATTR_CHNF_REGHANDLE of the statement handle, OCI_HTYPE_STMT. The registration is completed when the query is executed.

   See Also: "OCI_ATTR_CHNF_REGHANDLE" on page A-27
9. Optionally unregister a subscription. The client can call the OCISubscriptionUnRegister() function with the subscription handle as a parameter.

A binding of a statement handle to a subscription handle is only valid for the first execution of a query. If the application must use the same OCI statement handle for subsequent executions, it must repopulate the registration handle attribute of the statement handle. A binding of a subscription handle to a statement handle is only permitted when the statement is a query (determined at execute time). If a DML statement is executed as part of the execution, then an exception is issued.

**Subscription Handle Attributes for Continuous Query Notification**

The subscription handle attributes for continuous query notification can be divided into generic attributes (common to all subscriptions) and namespace-specific attributes (particular to continuous query notification).
The WRITE attributes on the statement handle can only be modified before the registration is created.

**Generic Attributes - Common to All Subscriptions**

OCI_ATTR_SUBSCR_NAMESPACE (WRITE) - Set this attribute to OCI_SUBSCR_NAMESPACE_DBCHANGE for subscription handles.

OCI_ATTR_SUBSCR_CALLBACK (WRITE) - Use this attribute to store the callback associated with the subscription handle. The callback is executed when a notification is received.

When a new continuous query notification message becomes available, the callback is invoked in the listener thread with desc pointing to a descriptor of type OCI_DTYPE_CHDES that contains detailed information about the invalidation.

OCI_ATTR_SUBSCR_QOSFLAGS - This attribute is a generic flag with the following values:

```c
#define OCI_SUBSCR_QOS_RELIABLE 0x01 /* reliable */
#define OCI_SUBSCR_QOS_PURGE_ON_NTFN 0x10 /* purge on first ntfn */
```

- **OCI_SUBSCR_QOS_RELIABLE** - Set this bit to allow notifications to be persistent. Therefore, you can use surviving instances of an Oracle RAC cluster to send and retrieve invalidation messages, even after a node failure, because invalidations associated with this registration ID are queued persistently into the database. If this bit is FALSE, then invalidations are enqueued in to 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_SUBSCR_QOS_PURGE_ON_NTFN** - Set this bit to allow the registration to be purged on the first notification.

A parallel example is presented in "Publish-Subscribe Registration Functions in OCI" on page 9-55.

OCI_ATTR_SUBSCR_CQ_QOSFLAGS - This attribute describes the continuous query notification-specific QOS flags (mode is WRITE, data type is ub4), which are:

- **0x1 OCI_SUBSCR_CQ_QOS_QUERY** - Set this flag to indicate that query-level granularity is required. Notification should be only generated if the query result set changes. By default, this level of QOS has no false positives.

- **0x2 OCI_SUBSCR_CQ_QOS_BEST_EFFORT** - Set this flag to indicate that best effort filtering is acceptable. It may be used by caching applications. The database may use heuristics based on cost of evaluation and avoid full pruning in some cases.

OCI_ATTR_SUBSCR_TIMEOUT - Use this attribute to specify a ub4 timeout value defined in seconds. If the timeout value is 0 or not specified, then the registration is active until explicitly unregistered.

**Namespace- Specific or Feature-Specific Attributes**

The following attributes are namespace-specific or feature-specific to the continuous query notification feature.

OCI_ATTR_CHNF_TABLENAMES (data type is (OCIColl *)) - These attributes are provided to retrieve the list of table names that were registered. These attributes are available from the subscription handle, after the query is executed.

OCI_ATTR_CHNF_ROWIDS - A Boolean attribute (default FALSE). If set to TRUE, then the continuous query notification message includes row-level details such as operation type and ROWID.
OCI_ATTR_CHNF_OPERATIONS - Use this ub4 flag to selectively filter notifications based on operation type. This option is ignored if the registration is of query-level granularity. Flags stored are as follows:

- OCI_OPCODE_ALL - All operations
- OCI_OPCODE_INSERT - Insert operations on the table
- OCI_OPCODE_UPDATE - Update operations on the table
- OCI_OPCODE_DELETE - Delete operations on the table

OCI_ATTR_CHNF_CHANGELAG - The client can use this ub4 value to specify the number of transactions by which the client is willing to lag behind. The client can also use this option as a throttling mechanism for continuous query notification messages. When you choose this option, ROWID-level granularity of information is not available in the notifications, even if OCI_ATTR_CHNF_ROWIDS was set to TRUE. This option is ignored if the registration is of query-level granularity.

Once the OCISubscriptionRegister() call is invoked, none of the preceding attributes (generic, name-specific, or feature-specific) can be modified on the registration already created. Any attempt to modify those attributes is not reflected on the registration already created, but it does take effect on newly created registrations that use the same registration handle.

See Also: "Continuous Query Notification Attributes" on page A-57

Notifications can be spaced out by using the grouping NTFN option. The relevant generic notification attributes are:

OCI_ATTR_SUBSCR_NTFN_GROUPING_VALUE
OCI_ATTR_SUBSCR_NTFN_GROUPING_TYPE
OCI_ATTR_SUBSCR_NTFN_GROUPING_START_TIME
OCI_ATTR_SUBSCR_NTFN_GROUPING_REPEAT_COUNT

See Also: "Publish-Subscribe Register Directly to the Database" on page 9-55 for more details about these attributes

Using OCI_ATTR_CQ_QUERYID Attribute

The attribute OCI_ATTR_CQ_QUERYID on the statement handle, OCI_HTYPE_STMT, obtains the query ID of a registered query after registration is made by the call to OCIStmtExecute().

See Also: "OCI_ATTR_CQ_QUERYID" on page A-28

Continuous Query Notification Descriptors

The continuous query notification descriptor is passed into the desc parameter of the notification callback specified by the application. The following attributes are specific to continuous query notification. The OCI type constant of the continuous query notification descriptor is OCI_DTYPE_CHDES.

The notification callback receives the top-level notification descriptor, OCI_DTYPE_CHDES, as an argument. This descriptor in turn includes either a collection of OCI_DTYPE_CQDES or OCI_DTYPE_TABLE_CHDES descriptors based on whether the event type was OCI_EVENT_QUERYCHANGE or OCI_EVENT_OBJCHANGE. An array of table continuous query descriptors is embedded inside the continuous query descriptor for notifications of type OCI_EVENT_QUERYCHANGE. If ROWID level granularity of information was
requested, each `OCI_DTYPE_TABLE_CHDES` contains an array of row-level continuous query descriptors (`OCI_DTYPE_ROW_CHDES`) corresponding to each modified `ROWID`.

**OCI_DTYPE_CHDES**

This is the top-level continuous query notification descriptor type.

- `OCI_ATTR_CHDES_DBNAME (oratext *)` - Name of the database (source of the continuous query notification)
- `OCI_ATTR_CHDES_XID (RAW(8))` - Message ID of the message

**OCI_ATTR_CHDES_NFYTYPE** - Flags describing the notification type:

- 0x0 `OCI_EVENT_NONE` - No further information about the continuous query notification
- 0x1 `OCI_EVENT_STARTUP` - Instance startup
- 0x2 `OCI_EVENT_SHUTDOWN` - Instance shutdown
- 0x3 `OCI_EVENT_SHUTDOWN_ANY` - Any instance shutdown - Oracle Real Application Clusters (Oracle RAC)
- 0x5 `OCI_EVENT_DEREG` - Unregistered or timed out
- 0x6 `OCI_EVENT_OBJCHANGE` - Object change notification
- 0x7 `OCI_EVENT_QUERYCHANGE` - Query change notification

**OCI_ATTR_CHDES_TABLE_CHANGES** - A collection type describing operations on tables of data type (`OCIColl *`). This attribute is present only if the `OCI_ATTR_CHDES_NFTYPE` attribute was of type `OCI_EVENT_OBJCHANGE`; otherwise, it is `NULL`. Each element of the collection is a table of continuous query descriptors of type `OCI_DTYPE_TABLE_CHDES`.

**OCI_ATTR_CHDES_QUERIES** - A collection type describing the queries that were invalidated. Each member of the collection is of type `OCI_DTYPE_CQDES`. This attribute is present only if the attribute `OCI_ATTR_CHDES_NFTYPE` was of type `OCI_EVENT_QUERYCHANGE`; otherwise, it is `NULL`.

**OCI_DTYPE_CQDES** This notification descriptor describes a query that was invalidated, usually in response to the commit of a DML or a DDL transaction. It has the following attributes:

- `OCI_ATTR_CQDES_OPERATION (ub4, READ)` - Operation that occurred on the query. It can be one of these values:
  - `OCI_EVENT_QUERYCHANGE` - Query result set change
  - `OCI_EVENT_DEREG` - Query unregistered
- `OCI_ATTR_CQDES_TABLE CHANGES (OCIColl *, READ)` - 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`.
- `OCI_ATTR_CQDES_QUERYID (ub8, READ)` - Query ID of the query that was invalidated.

**OCI_DTYPE_TABLE_CHDES** This notification descriptor conveys information about changes to a table involved in a registered query. It has the following attributes:

- `OCI_ATTR_CHDES_TABLE_NAME (oratext *)` - Schema annotated table name.
- `OCI_ATTR_CHDES_TABLE_OPFLAGS (ub4)` - Flag field describing the operations on the table. Each of the following flag fields is in a separate bit position in the attribute:
- 0x1 OCI_OPCODE_ALLROWS - The table is completely invalidated.
- 0x2 OCI_OPCODE_INSERT - Insert operations on the table.
- 0x4 OCI_OPCODE_UPDATE - Update operations on the table.
- 0x8 OCI_OPCODE_DELETE - Delete operations on the table.
- 0x10 OCI_OPCODE_ALTER - Table altered (schema change). This includes DDL statements and internal operations that cause row migration.
- 0x20 OCI_OPCODE_DROP - Table dropped.

OCI_ATTR_CHDES_TABLE_ROW_CHANGES - This is an embedded collection describing the changes to the rows within the table. Each element of the collection is a row continuous query descriptor of type OCI_DTYPE_ROW_CHDES that has the following attributes:

- OCI_ATTR_CHDES_ROW_ROWID (OraText *) - String representation of a ROWID.
- OCI_ATTR_CHDES_ROW_OPFLAGS - Reflects the operation type: INSERT, UPDATE, DELETE, or OTHER.

See Also: "Continuous Query Notification Descriptor Attributes" on page A-58

Continuous Query Notification Example

Example 10–1 is a simple OCI program, demoquery.c. See the comments in the listing. The calling session must have the CHANGE NOTIFICATION system privilege and SELECT privileges on all objects that it attempts to register.

Example 10–1 Program Listing That Demonstrates Continuous Query Notification

/* Copyright (c) 2010, Oracle. All rights reserved. */

#ifndef S_ORACLE
# include <oratypes.h>
#endif

/**************************************************************************
*This is a DEMO program. To test, compile the file to generate the executable
*demoquery. Then demoquery can be invoked from a command prompt.
*It will have the following output:

Initializing OCI Process
Registering query : select last_name, employees.department_id, department_name from employees, departments where employee_id = 200 and employees.department_id = departments.department_id
Query Id 23
Waiting for Notifications

*Then from another session, log in as HR/HR and perform the following
*DML transactions. It will cause two notifications to be generated.

update departments set department_name = 'Global Admin' where department_id=10;
commit;
update departments set department_name = 'Adminstration' where department_id=10;
commit;

*The demoquery program will now show the following output corresponding
*to the notifications received.

Query 23 is changed
Table changed is HR.DEPARTMENTS table_op 4
Row changed is AAAMBoAABAAAAKX2AAA row_op 4
Query 23 is changed
Table changed is HR.DEPARTMENTS table_op 4
Row changed is AAAMBoAABAAAAKX2AAA row_op 4

*The demo program waits for exactly 10 notifications to be received before
*logging off and unregistering the subscription.

**************************************************************************/

/*---------------------------------------------------------------------------
PRIVATE TYPES AND CONSTANTS
---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------
STATIC FUNCTION DECLARATIONS
---------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>
#define MAXSTRLENGTH 1024
#define bit(a,b) ((a)&(b))

static int notifications_processed = 0;
static OCISubscription *subhandle1 = (OCISubscription *)0;
static OCISubscription *subhandle2 = (OCISubscription *)0;
static void checker(/*_ OCIError *errhp, sword status _*/);
static void registerQuery(/*_ OCISvcCtx *svchp, OCIError *errhp, OCIStmt *stmthp,
OCIEnv *envhp _*/);
static void myCallback (/*_ dvoid *ctx, OCISubscription *subscrhp,
dvoid *payload, ub4 *payl, dvoid *descriptor,
ub4 mode _*/);
static int NotificationDriver(/*_ int argc, char *argv[]  _*/);
static sword status;
static boolean logged_on = FALSE;

int main(int argc, char **argv)
{
    NotificationDriver(argc, argv);
    return 0;
}
int NotificationDriver(argc, argv)
int argc;
char *argv[];
{
    OCIEnv *envhp;
    OCISvcCtx *svchp, *svchp2;
    OCIError *errhp, *errhp2;
    OCISession *authp, *authp2;
    OCISession *authp, *authp2;
    OCISession *authp, *authp2;
    OCISession *authp, *authp2;
    OCISession *authp, *authp2;
    OCIStmt *stmthp, *stmthp2;
    OCIDuration dur, dur2;
    int i;
dvoid *tmp;
    OCISession *usrhp;
    OCIServer *srvhp;

    printf("Initializing OCI Process\n");
    /* Initialize the environment. The environment must be initialized
        with OCI_EVENTS and OCI_OBJECT to create a continuous query notification
        registration and receive notifications.
        */
    OCIEnvCreate( (OCIEnv **) &envhp, OCI_EVENTS | OCI_OBJECT, (dvoid *)0,
        (void (*)(dvoid *, size_t)) 0,
        (void (*)(dvoid *, char *, size_t))0,
        (void (*)(dvoid *, dvoid *)) 0,
        (size_t) 0, (dvoid **) 0 );
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
        (size_t) 0, (dvoid **) 0);
    /* server contexts */
    OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
        (size_t) 0, (dvoid **) 0);
    OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
        (size_t) 0, (dvoid **) 0);
    checker(errhp,OCIServerAttach(srvhp, errhp, (text *) 0, (ub4) 0,
        (ub4) OCI_DEFAULT));
    /* set attribute server context in the service context */
    OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp,
        (ub4) 0, (ub4) OCI_ATTR_SESSION, (OCIError *) errhp);

    /* allocate a user context handle */
    OCIHandleAlloc((dvoid *) envhp, (dvoid **) &usrhp, (ub4) OCI_HTYPE_SESSION,
        (size_t) 0, (dvoid **) 0);
    OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
        (text *)"HR", (ub4)strlen((char *)"HR"),
        OCI_ATTR_USERNAME, errhp);
    OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
        (text *)"HR", (ub4)strlen((char *)"HR"),
        OCI_ATTR_PASSWORD, errhp);
    checker(errhp,OCISessionBegin (svchp, errhp, ushrp, OCI_CRED_RDBMS,
        OCI_DEFAULT));
    /* Allocate a statement handle */
    OCIHandleAlloc((dvoid *) envhp, (dvoid **) &stmthp,
        (ub4) OCI_HTYPE_STMT, 52, (dvoid **) &tmp);
    OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *) ushrp, (ub4)0,
        OCI_ATTR_SESSION, errhp);
registerQuery(svchp, errhp, stmthp, envhp);
printf("Waiting for Notifications\n");
while (notifications_processed !=10)
{
    sleep(1);
}
printf("Going to unregister HR\n");
fflush(stdout);
/* Unregister HR */
checker(errhp,
    OCISubscriptionUnRegister(svchp, subhandle1, errhp, OCI_DEFAULT));
checker(errhp, OCIAbort(svchp, errhp, OCI_DEFAULT));
printf("HR Logged off.\n");
if (subhandle1)
    OCIHandleFree((dvoid *)subhandle1, OCI_HTYPE_SUBSCRIPTION);
if (stmthp)
    OCIHandleFree((dvoid *)stmthp, OCI_HTYPE_STMT);
if (srvhp)
    OCIHandleFree((dvoid *)srvhp, (ub4) OCI_HTYPE_SERVER);
if (svchp)
    OCIHandleFree((dvoid *)svchp, (ub4) OCI_HTYPE_SVCCTX);
if (authp)
    OCIHandleFree((dvoid *)authp, (ub4) OCI_HTYPE_SESSION);
if (errhp)
    OCIHandleFree((dvoid *)errhp, (ub4) OCI_HTYPE_ERROR);
if (envhp)
    OCIHandleFree((dvoid *)envhp, (ub4) OCI_HTYPE_ENV);

return 0;
}

void checker(errhp, status)
    OCIError *errhp;
    sword status;
{
    text errbuf[512];
    sb4 errcode = 0;
    int retval = 1;

    switch (status)
    {
    case OCI_SUCCESS:
        retval = 0;
        break;
    case OCI_SUCCESS_WITH_INFO:
        (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
        break;
    case OCI_NEED_DATA:
        (void) printf("Error - OCI_NEED_DATA\n");
        break;
    case OCI_NO_DATA:
        (void) printf("Error - OCI_NO_DATA\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;
    }
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;
}
if (retval) {
    exit(1);
}

void processRowChanges(OCIEnv *envhp, OCIError *errhp, OCIStmt *stmthp,
    OCIColl *row_changes)
{
    dvoid **row_descp;
    dvoid *row_desc;
    boolean exist;
    ub2 i, j;
    dvoid *elemind = (dvoid *)0;
    oratext *row_id;
    ub4 row_op;

    sb4 num_rows;
    if (!row_changes) return;
    checker(errhp, OCICollSize(envhp, errhp,
        (CONST OCIColl *) row_changes, &num_rows));
    for (i=0; i<num_rows; i++) {
        checker(errhp, OCICollGetElem(envhp,
            errhp, (OCIColl *) row_changes,
            i, &exist, &row_descp, &elemind));
        row_desc = *row_descp;
        checker(errhp, OCIAttrGet (row_desc,
            OCI_DTYPE_ROW_CHDES, (dvoid *)&row_id,
            NULL, OCI_ATTR_CHDES_ROW_ROWID, errhp));
        checker(errhp, OCIAttrGet (row_desc,
            OCI_DTYPE_ROW_CHDES, (dvoid *)&row_op,
            NULL, OCI_ATTR_CHDES_ROW_OPFLAGS, errhp));
        printf("Row changed is %s row_op %d\n", row_id, row_op);
        fflush(stdout);
    }
}

void processTableChanges(OCIEnv *envhp, OCIError *errhp, OCIStmt *stmthp,
    OCIColl *table_changes)
{
    dvoid **table_descp;
    dvoid *table_desc;

    sb4 num_rows;
    if (!table_changes) return;
    checker(errhp, OCICollSize(envhp, errhp,
        (CONST OCIColl *) table_changes, &num_rows));
    for (i=0; i<num_rows; i++) {
        checker(errhp, OCICollGetElem(envhp,
            errhp, (OCIColl *) table_changes,
            i, &exist, &table_descp, &elemind));
        table_desc = *table_descp;
        checker(errhp, OCIAttrGet (table_desc,
            OCI_DTYPE_TAB_CHDES, (dvoid *)&table_id,
            NULL, OCI_ATTR_CHDES_TAB_TABID, errhp));
        checker(errhp, OCIAttrGet (table_desc,
            OCI_DTYPE_TAB_CHDES, (dvoid *)&table_op,
            NULL, OCI_ATTR_CHDES_TAB_OPFLAGS, errhp));
        printf("Table changed is %s table_op %d\n", table_id, table_op);
        fflush(stdout);
    }
}
Continuous Query Notification

dvoid **row_descp;
dvoid *row_descp;
OCIColl *row_changes = (OCIColl *)0;
boolean exist;
ub2 i, j;
dvoid *elemind = (dvoid *)0;
oratext *table_name;
ub4 table_op;
ub4 num_tables;
if (!table_changes) return;
checker(errhp, OCICollSize(envhp, errhp,
(CONST OCIColl *) table_changes, &num_tables));
for (i=0; i<num_tables; i++)
{
  checker(errhp, OCICollGetElem(envhp,
    errhp, (OCIColl *) table_changes,
    i, &exist, &table_descp, &elemind));
  table_desc = *table_descp;
  checker(errhp, OCIAttrGet (table_desc,
    OCI_DTYPE_TABLE_CHDES, (dvoid *)&table_name,
    NULL, OCI_ATTR_CHDES_TABLE_NAME, errhp));
  checker(errhp, OCIAttrGet (table_desc,
    OCI_DTYPE_TABLE_CHDES, (dvoid *)&table_op,
    NULL, OCI_ATTR_CHDES_TABLE_OPFLAGS, errhp));
  checker(errhp, OCIAttrGet (table_desc,
    OCI_DTYPE_TABLE_CHDES, (dvoid *)&row_changes,
    NULL, OCI_ATTR_CHDES_TABLE_ROW_CHANGES, errhp));
  printf ("Table changed is %s table_op %d\n", table_name,table_op);
  fflush(stdout);
  if (!bit(table_op, OCI_OPCODE_ALLROWS))
    processRowChanges(envhp, errhp, stmthp, row_changes);
}

void processQueryChanges(OCIEnv *envhp, OCIError *errhp, OCIStmt *stmthp,
  OCIColl *query_changes)
{
  sb4 num_queries;
  ub8 queryid;
  OCINumber qidnum;
  ub4 queryop;
dvoid *elemind = (dvoid *)0;
dvoid *query_desc;
dvoid **query_descp;
ub2 i;
boolean exist;
OCIColl *table_changes = (OCIColl *)0;
if (!query_changes) return;
checker(errhp, OCICollSize(envhp, errhp,
(CONST OCIColl *) query_changes, &num_queries));
for (i=0; i < num_queries; i++)
{
  checker(errhp, OCICollGetElem(envhp,
    errhp, (OCIColl *) query_changes,
    i, &exist, &query_descp, &elemind));
Continuous Query Notification

```c
query_desc = *query_descp;
checker(errhp, OCIAttrGet (query_desc,
    OCI_DTYPE_CQDES, (dvoid *)&queryid,
    NULL, OCI_ATTR_CQDES_QUERYID, errhp));
checker(errhp, OCIAttrGet (query_desc,
    OCI_DTYPE_CQDES, (dvoid *)&queryop,
    NULL, OCI_ATTR_CQDES_OPERATION, errhp));
printf(' Query %d is changed\n", queryid);
if (queryop == OCI_EVENT_DEREG)
    printf('Query Deregistered\n");
checker(errhp, OCIAttrGet (query_desc,
    OCI_DTYPE_CQDES, (dvoid *)&table_changes,
    NULL, OCI_ATTR_CQDES_TABLE_CHANGES, errhp));
    processTableChanges(envhp, errhp, stmthp, table_changes);
}
}

void myCallback (ctx, subscrhp, payload, payl, descriptor, mode)
    dvoid *ctx;
    OCISubscription *subscrhp;
    dvoid *payload;
    ub4 *payl;
    dvoid *descriptor;
    ub4 mode;
    {
        OCIColl *table_changes = (OCIColl *)0;
        OCIColl *row_changes = (OCIColl *)0;
        dvoid *change_descriptor = descriptor;
        ub4 notify_type;
        ub2 i, j;
        OCIEnv *envhp;
        OCIError *errhp;
        OCIColl *query_changes = (OCIColl *)0;
        OCIServer *srvhp;
        OCISvcCtx *svchp;
        OCISession *usrhp;
        dvoid     *tmp;
        OCIStmt *stmthp;

        (void)OCIEnvInit( (OCIEnv **) &envhp, OCI_DEFAULT, (size_t)0, (dvoid **)0 );
        (void) OCIClientAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                                  (size_t) 0, (dvoid **) 0);
        /* server contexts */
        (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                                 (size_t) 0, (dvoid **) 0);
        (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                                 (size_t) 0, (dvoid **) 0);

        OCIAttrGet (change_descriptor, OCI_DTYPE_CHDES, (dvoid *) &notify_type,
                                  NULL, OCI_ATTR_CHDES_NFYTYPE, errhp);
        fflush(stdout);
        if (notify_type == OCI_EVENT_SHUTDOWN ||
            notify_type == OCI_EVENT_SHUTDOWN_ANY) {
            printf(' Query %d is changed\n", notify_type);
            printf('Query Deregistered\n");
            OCIAttrGet (query_desc, OCI_DTYPE_CQDES, (dvoid *)&queryid,
                                  NULL, OCI_ATTR_CQDES_QUERYID, errhp));
            printf(' Query %d is changed\n", notify_type);
            printf('Query Deregistered\n");
        }
    }
```
printf("SHUTDOWN NOTIFICATION RECEIVED\n");
fflush(stdout);
notifications_processed++; return;
}

if (notify_type == OCI_EVENT_STARTUP)
{
 printf("STARTUP NOTIFICATION RECEIVED\n");
fflush(stdout);
notifications_processed++; return;
}

notifications_processed++;
checker(errhp, OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT));

OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX, 52, (dvoid **) &tmp);
/* set attribute server context in the service context */
OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp,
(ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

/* allocate a user context handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **) &usrhp, (ub4) OCI_HTYPE_SESSION,
(size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
(dvoid *)'HR', (ub4)strlen("HR"), OCI_ATTR_USERNAME, errhp);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
(dvoid *)'HR', (ub4)strlen("HR"),
OCI_ATTR_PASSWORD, errhp);

checker(errhp, OCISessionBegin(svchp, errhp, usrhp, OCI_CRED_RDBMS,
OCI_DEFAULT));

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
(dvoid *)0, OCI_ATTR_SESSION, errhp);

/* Allocate a statement handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **) &stmthp,
(ub4) OCI_HTYPE_STMT, 52, (dvoid **) &tmp);

if (notify_type == OCI_EVENT_OBJCHANGE)
{
 checker(errhp, OCIAttrGet (change_descriptor,
 OCI_DTYPE_CHDES, &table_changes, NULL,
 OCI_ATTR_CHDES_TABLE_CHANGES, errhp));
 processTableChanges(envhp, errhp, stmthp, table_changes);
}
else if (notify_type == OCI_EVENT_QUERYCHANGE)
{
 checker(errhp, OCIAttrGet (change_descriptor,
 OCI_DTYPE_CHDES, &query_changes, NULL,
 OCI_ATTR_CHDES_QUERIES, errhp));
 processQueryChanges(envhp, errhp, stmthp, query_changes);
}
checker(errhp, OCISessionEnd(svchp, errhp, usrhp, OCI_DEFAULT));
checker(errhp, OCIServerDetach(srvhp, errhp, OCI_DEFAULT));
Continuous Query Notification

if (stmthp)
    OCIHandleFree((dvoid *)stmthp, OCI_HTYPE_STMT);
if (errhp)
    OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR);
if (srvhp)
    OCIHandleFree((dvoid *)srvhp, OCI_HTYPE_SERVER);
if (svchp)
    OCIHandleFree((dvoid *)svchp, OCI_HTYPE_SVCCTX);
if (usrhp)
    OCIHandleFree((dvoid *)usrhp, OCI_HTYPE_SESSION);
if (envhp)
    OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);

}  

void registerQuery(svchp, errhp, stmthp, envhp)
    OCISvcCtx *svchp;
    OCIError *errhp;
    OCIStmt *stmthp;
    OCIEnv *envhp;
    {
        OCISubscription *subscrhp;
        ub4 namespace = OCI_SUBSCR_NAMESPACE_DBCHANGE;
        ub4 timeout = 60;
        OCIDefine *defnp1 = (OCIDefine *)0;
        OCIDefine *defnp2 = (OCIDefine *)0;
        OCIDefine *defnp3 = (OCIDefine *)0;
        OCIDefine *defnp4 = (OCIDefine *)0;
        OCIDefine *defnp5 = (OCIDefine *)0;
        int mgr_id =0;
        text query_text1[] = "select last_name, employees.department_id, department_name \nfrom employees,departments where employee_id = 200 and employees.department_id =\ndepartments.department_id";
        ub4 num_prefetch_rows = 0;
        ub4 num_reg_tables;
        OCIColl *table_names;
        ub2 i;
        boolean rowids = TRUE;
        ub4 qosflags = OCI_SUBSCR_CQ_QOS_QUERY  ;
        int empno=0;
        OCINumber qidnum;
        ub8 qid;
        char outstr[MAXSTRLENGTH], dname[MAXSTRLENGTH];
        int q3out;

        fflush(stdout);
        /* allocate subscription handle */
        OCIHandleAlloc ((dvoid *) envhp, (dvoid **) &subscrhp,
            OCI_HTYPE_SUBSCRIPTION, {size_t} 0, (dvoid **) 0);

        /* set the namespace to DBCHANGE */
        checker(errhp, OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
            {void *__)namespace, sizeof(ub4),
            OCI_ATTR_SUBSCR_NAMESPACE, errhp));

        /* Associate a notification callback with the subscription */
        checker(errhp, OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
            {void *__)myCallback, 0, OCI_ATTR_SUBSCR_CALLBACK, errhp));
        /* Allow extraction of rowid information */

More OCI Advanced Topics  10-15
checker(errhp, OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
    (dvoid *)&rowids, sizeof(ub4),
    OCI_ATTR_CHNF_ROWIDS, errhp));

checker(errhp, OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
    (dvoid *)&qosflags, sizeof(ub4),
    OCI_ATTR_SUBSCR_CQ_QOSFLAGS, errhp));

/* Create a new registration in the DBCHANGE namespace */
checker(errhp, 
    OCISubscriptionRegister(svchp, &subscrhp, 1, errhp, OCI_DEFAULT));

/* Multiple queries can now be associated with the subscription */

subhandle1 = subscrhp;

printf("Registering query : %s\n", (const signed char *)query_text1);
/* Prepare the statement */
checker(errhp, OCIStmtPrepare (stmthp, errhp, query_text1,
    (ub4)strlen((const signed char *)query_text1), OCI_V7_SYNTAX,
    OCI_DEFAULT));

checker(errhp,
    OCIDefineByPos(stmthp, &defnp1, 
        (dvoid *)outstr, MAXSTRLENGTH * sizeof(char),
        SQLT_STR, (dvoid *)0, (ub2 *)0, (ub2 *)0, OCI_DEFAULT));

checker(errhp,
    OCIDefineByPos(stmthp, &defnp2, 
        errhp, 2, (dvoid *)&empno, sizeof(empno),
        SQLT_INT, (dvoid *)0, (ub2 *)0, (ub2 *)0, OCI_DEFAULT));

checker(errhp,
    OCIDefineByPos(stmthp, &defnp3, 
        errhp, 3, (dvoid *)&dname, sizeof(dname),
        SQLT_STR, (dvoid *)0, (ub2 *)0, (ub2 *)0, OCI_DEFAULT));

/* Associate the statement with the subscription handle */
OCIAttrSet (stmthp, OCI_HTYPE_STMT, subscrhp, 0,
    OCI_ATTR_CHNF_REGHANDLE, errhp);

/* Execute the statement, the execution performs object registration */
checker(errhp, OCIStmtExecute (svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
    (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL ,
    OCI_DEFAULT));

fflush(stdout);

OCIAttrGet(stmthp, OCI_HTYPE_STMT, &qid, (ub4 *)0,
    OCI_ATTR_CQ_QUERYID, errhp);
printf("Query Id %d\n", qid);

/* commit */
checker(errhp, OCITransCommit(svchp, errhp, (ub4) 0));

static void cleanup(envhp, svchp, srvhp, errhp, usrhp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIServer *srvhp;
OCIError *errhp;
OCIContext *usrhp;
{
  /* detach from the server */
  checker(errhp, OCIContextEnd(svchp, errhp, usrhp, OCI_DEFAULT));
  checker(errhp, OCIContextDetach(srvhp, errhp, (ub4)OCI_DEFAULT));

  if (usrhp)
    (void) OCIHandleFree((dvoid *) usrhp, (ub4) OCI_HTYPE_SESSION);
  if (svchp)
    (void) OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX);
  if (srvhp)
    (void) OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER);
  if (errhp)
    (void) OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR);
  if (envhp)
    (void) OCIHandleFree((dvoid *) envhp, (ub4) OCI_HTYPE_ENV);
}

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()" on page 16-12
- "OCIDBShutdown()" on page 16-10
- "Administration Handle Attributes" on page A-22
- Oracle Database Administrator’s Guide
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 10–2 shows sample code that uses a client-side parameter file (pfile) that is set in the administration handle and performs a database startup operation.

Example 10–2  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, OCIHHandleAlloc((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, OCICStmtPrepare2(svchp, &stmthp, errhp, mount_stmt, (ub4)
   strlen((char*) mount_stmt), 
   (CONST OraText *) 0, (ub4) 0, (ub4) OCI_NTV_SYNTAX, (ub4) 
   OCI_DEFAULT));
checkerr(errhp, OCICStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4)0, 
   (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT));
checkerr(errhp, OCICStmtRelease(stmthp, errhp, (OraText *)0, 0, OCI_DEFAULT));

/* Open the database */
checkerr(errhp, OCICStmtPrepare2(svchp, &stmthp, errhp, open_stmt, (ub4)
   strlen((char*) open_stmt), 
   (CONST OraText *)0, (ub4)0, (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4)0,
  (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT));
checkerr(errhp, OCIStmtRelease(stmthp, errhp, (OraText *)0, 0, OCI_DEFAULT));

/* End the sysdba session */
OCISessionEnd(svchp, errhp, usrhp, (ub4)OCI_DEFAULT);
...

To 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 10–3 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 10–3 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_DEFAULT));
...
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 ROWIDs 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_AVF (CHAR2)
- 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 ROWIDs (for example, if getting it in SQLT_STR format), allocate enough memory for storing ROWIDs. Remember the differences between the ROWID data type and the UROWID data type. The ROWID data type can only store physical ROWIDs, but UROWID can store logical ROWIDs (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() 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 ROWIDs, in this way:

OCIAttrSet(stmthp, OCI_HTYPE_STMT, 0, 0, OCI_ATTR_FETCH_ROWID, errhp);

Dynamic define is not compatible with implicit fetching of ROWIDs. 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() for position 0, is for fetching an array of data
simultaneously into the user buffers and getting their respective ROWIDs at the same
time. It allows for fetching of ROWIDs 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 ROWIDs, the attribute OCI_ATTR_ROWID on the
statement handle cannot be used simultaneously to get the ROWIDs. You can only use
one of them at a time for a particular statement handle.

See Also:  "OCI_ATTR_FETCH_ROWID" on page A-28

Example of Implicit Fetching of ROWIDs

Use the fragment of a C program in Example 10–5 to build upon.

Example 10–5  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 (OCPStmtPrepare (select_p, errhp,
    mySql, strlen(mySql), OCI_NTVC_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]);
Client Result Cache

OCI applications can use client memory to take advantage of the OCI result cache to improve response times of repetitive queries.

The client result cache enables client-side caching of SQL query result sets in client memory. The OCI result cache is completely transparent to OCI applications, and its cache of result set data is kept consistent with any session or database changes that affect its result set.

Applications employing this feature see improved performance for queries that have a cache hit. OCI can transparently use cached results for future executions of these queries. Because retrieving results locally from an OCI client process is faster than making a database call and rerunning a query, frequently run queries experience a significant performance improvement when their results are cached.

The OCI cache also reduces the server CPU that would have been consumed for processing the query, thereby improving server scalability. OCI statements from multiple sessions can match the same cached result set in the OCI process memory, if they have similar schema, SQL text, bind values, and session settings. Otherwise, with any dissimilarity, the query execution is directed to the server.

You must enable OCI statement caching or cache statements at the application level when using the client result cache.

Client result cache works with OCI features such as OCI session pooling, OCI connection pooling, database resident connection pooling, and OCI transparent application failover (TAF).

See Also: “Statement Caching in OCI” on page 9-26

Benefits of Client Result Cache

The benefits of OCI client query result cache are as follows:

- Because the result cache is on the client side, a cache hit causes OCIStmtExecute() and OCIStmtFetch2() calls to be processed locally, instead of making server round-trips. This can result in huge performance savings for server resources, for example, server CPU and server I/O.

- The OCI client-side query result set cache is a transparent and consistent cache.

- The result cache on OCI client is per-process, so multiple client sessions can simultaneously use matching cached result sets.

- It minimizes the need for each OCI application to have its own custom result set cache.

- It transparently manages the caching aspects of the cached result sets, that is: concurrent access by multiple threads, multiple statements, multiple sessions, invalidation, refreshing of result sets in the cache, and cache memory management.

- It transparently invalidates the cached result sets on any database changes that may affect the result sets, when an OCI process makes round-trips to the server.
This consistent cache is automatically available to all OCI applications and drivers (such as JDBC OCI, ODP.Net, OCCI, Pro*C/C++, Pro*COBOL, ODBC, and so on) built using OCI.

- The cache uses OCI client memory that may be less expensive than server memory.
- A local cache on the client has better locality of reference for queries executed by that client.

**Guidelines for Using Client Result Cache**

You can enable client result caching in several ways for your application and establish an order of precedence in its usage based on the methods selected. See "Cache Example Use Cases" on page 10-27 for more usage information.

- **SQL Hints** - Annotate a query with a SQL hint /*+ result_cache */ to indicate that results are to be stored in the query result cache. Using SQL hints is the highest order of precedence; it takes precedence over table annotations and session parameters. It is applicable to a single query. This method requires application-level changes.

- **Table Annotation** - Annotate a table during deployment using result cache hints in the `ALTER TABLE` and `CREATE TABLE` statements. Using table annotation is the next highest order of precedence below SQL hints and above session parameters when using `MODE FORCE`. It is applicable to all queries for that table. This method requires no application-level changes.

- **Session Parameters** - Works across all tables for all queries; use this method when possible. You can either set the `RESULT_CACHE_MODE` initialization parameter in the server parameter file (`init.ora`) or use `RESULT_CACHE_MODE` clause in the `ALTER SESSION` and the `ALTER SYSTEM` statements. Using session parameters is the lowest order of precedence; both SQL hints and table annotations take precedence over session parameters usage. It is the most widely effective usage being applicable to all tables. This method requires no application-level changes.

Oracle recommends that applications annotate tables and queries with result cache hints for read-only or read-mostly database objects. If the result caching happens for queries with large results, these results can use a large amount of cache memory.

As each set of bind values specified by the application creates a different cached result set (for the same SQL text), these result sets together can use a large amount of client result cache memory.

When client result caching is enabled, the query result set can be cached on the client or on the server or both. The client result caching can be enabled even if the server result cache (that is enabled by default) is disabled.

The first `OCIStmtExecute()` call of every OCI statement handle call always goes to the server even if there might be a valid cached result set. It is necessary that an `OCIStmtExecute()` call be made for each statement handle to be able to match a cached result set. Oracle recommends that applications have their own statement caching for OCI statement handles, or use OCI statement caching so that `OCIStmtPrepare2()` can return an OCI statement handle that has been executed once. Multiple OCI statement handles, from the same or different sessions, can simultaneously fetch data from the same cached result set.

For a result set to be cached, the `OCIStmtExecute()` or `OCIStmtFetch2()` calls that transparently create this cached result set must fetch rows until an `ORA-01403 *No Data Found*` error is returned. Subsequent `OCIStmtExecute()` or `OCIStmtFetch2()` calls
that match a locally cached result set need not fetch to completion.

**SQL Hints**

Unless the `RESULT_CACHE_MODE` server initialization parameter is set to `FORCE`, you must explicitly specify the queries to be cached using SQL hints. The SQL `/*+ result_cache */` or `/*+ no_result_cache */` hint must be set in SQL text passed to `OCIStmtPrepare()` and `OCIStmtPrepare2()` calls.

**Table Annotation**

The `ALTER TABLE` and `CREATE TABLE` statements enable you to annotate tables with result cache mode. There are also session parameters as mentioned in a later section. The matrix of table annotations and session parameters dictates the effective result cache mode for queries on that table. Note that SQL hints override table annotations and session parameters. The syntax is:

```
CREATE|ALTER TABLE `{<schema>.}table` ... [RESULT_CACHE (MODE {FORCE|DEFAULT})]
```

Here is an example of `CREATE TABLE`. It defines the table columns:

```
CREATE TABLE foo (a NUMBER, b VARCHAR2(20)) RESULT_CACHE (MODE FORCE);
```

Here is an example of `ALTER TABLE`:

```
ALTER TABLE foo RESULT_CACHE (MODE DEFAULT);
```

This `ALTER TABLE` statement is used to annotate tables so that results of statements or query blocks (for server result cache) using these tables are stored in the result cache. If a given query has a SQL hint `/*+ result_cache */` or `/*+ no_result_cache */` or if the parameter `RESULT_CACHE_MODE` is set to `FORCE`, then the hint or session variable take precedence over the table annotation.

You should annotate all tables you want stored in the result cache. Then all SQL queries, whether single table selects or with joins, for these tables with cache hints, are considered for caching assuming they are cache-worthy.

**See Also:**

- Oracle Database SQL Language Reference for more information about `RESULT_CACHE` clause, SQL hints, `ALTER TABLE`, and `CREATE TABLE`
- Oracle Database Reference for more information about `RESULT_CACHE_MODE`

**Table 10–1** summarizes the result cache annotation mode values.

<table>
<thead>
<tr>
<th><strong>Table 10–1</strong></th>
<th><strong>DDL Table Result Cache Annotation Modes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mode Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>DEFAULT</td>
<td>The default value. Result caching is not determined at the table level. You can use this value to clear any table annotations.</td>
</tr>
<tr>
<td>FORCE</td>
<td>If all table names in the query have this setting, then the query is always considered for caching unless the <code>NO_RESULT_CACHE</code> hint is specified for the query. If one or more tables named in the query are set to <code>DEFAULT</code>, then the effective table annotation for that query is <code>DEFAULT</code>.</td>
</tr>
</tbody>
</table>
Checking Table Annotation Mode

The `RESULT_CACHE` column in the DBA views `DBA_TABLES`, `USER_TABLES`, and `ALL_TABLES` shows the result cache mode annotation for the table. If the table has not been annotated, it shows `DEFAULT`.

Suppose that table `emp` is annotated as `ALTER TABLE emp RESULT_CACHE (MODE FORCE)`. Then execute the following query in the session:

```
SELECT table_name, result_cache FROM user_tables
```

The output is as follows:

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>RESULT_CACHE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP</td>
<td>FORCE</td>
</tr>
<tr>
<td>FOO</td>
<td>DEFAULT</td>
</tr>
</tbody>
</table>

The output shows that table `FOO` either has not been annotated or has been annotated using the following statement:

```
ALTER TABLE foo RESULT_CACHE (MODE DEFAULT);
```

See Also: Oracle Database Reference for more information about the `RESULT_CACHE` column on these DBA views

Session Parameters

The `RESULT_CACHE_MODE` parameter enables you to decide result cache mode across tables in your queries. Use this clause in `ALTER SESSION` and `ALTER SYSTEM` statements, or inside the server parameter file (`init.ora`) to determine result caching.

See Also: Oracle Database Reference for more information about `RESULT_CACHE_MODE`

Effective Result Cache Table Mode

The SQL query level result cache hints take precedence over the session parameter `RESULT_CACHE_MODE` and result cache table annotations. In addition, table annotation `FORCE` takes precedence over the session parameter `MANUAL` as indicated in Table 10–2.

Table 10–2 compares modes (`MANUAL` and `FORCE`) for the session parameter `RESULT_CACHE_MODE` versus the comparable table annotation modes and shows the effective result cache mode.

<table>
<thead>
<tr>
<th>RESULT_CACHE_MODE Parameter</th>
<th>MANUAL</th>
<th>FORCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table Annotation = <code>FORCE</code></td>
<td>FORCE</td>
<td>FORCE</td>
</tr>
<tr>
<td>Table Annotation = <code>DEFAULT</code></td>
<td>MANUAL</td>
<td>FORCE</td>
</tr>
</tbody>
</table>

Note that when the effective mode is `FORCE`, then the actual caching depends on internal restrictions for client and server cache, query cache worthiness (for example, there is no `SYSDATE` in the query), and space available in the cache. This is similar to the SQL query hint `/*+ result_cache */` because it is just a hint. It does not imply that the query is actually cached. Recall that table annotation `DEFAULT` indicates that result caching is not determined at the table level and session parameter mode `MANUAL` indicates that the query must be annotated with a SQL hint for the hint to take precedence, so in effect these are equivalent methods for this setting.
Cache Example Use Cases

The following are some use cases that show when SQL hints take precedence over table annotations and session parameter.

- If the `emp` table is annotated as `ALTER TABLE emp RESULT_CACHE (MODE FORCE)` and the session parameter is not set, (it has its default value of `MANUAL`), this implies queries on table `emp` are considered for query caching.

- If in an example, the SQL query is `SELECT /*+ no_result_cache */ empno FROM emp`, the query is not cached. This is because SQL hints take precedence over table annotations and session parameter.

- If the `emp` table is not annotated or is annotated as `ALTER TABLE emp RESULT_CACHE (MODE DEFAULT)` and the session parameter is not set (it has a default value of `MANUAL`), this implies queries are not cached.

- If in an example, the SQL query has the hint `SELECT /*+ result_cache */ * FROM emp`, then this query is considered for query caching.

- If there is no table annotation and there is no SQL query hint, but the session or system parameter is set to `FORCE`, all queries on all tables are considered for query caching.

See Also: Oracle Database SQL Language Reference for more about caching

Queries That Are Not Cached

There are queries that are not cached on the OCI client even if the result cache hint is specified. Such queries may be cached on the database if the server result cache feature is enabled (see the discussion of the SQL query result cache in Oracle Database Concepts for more information). If a SQL query includes any of the following, then the result set of that query is not cached in the OCI client result cache:

- Remote objects
- Complex types in the select list
- Snapshot-based queries or flashback queries
- Queries executed in a serializable, read-only transaction, or inside a flashback session
- Queries that have PL/SQL functions in them
- Queries that have virtual private database (VPD) policies enabled on the tables

Client Cache Consistency

The client cache transparently keeps the result set consistent with any session state or database changes that can affect its cached result sets.

When a transaction modifies the data or metadata of any of the database objects used to construct that cached result, invalidation is sent to the OCI client on its subsequent round-trip to the server. If the OCI application does no database calls for a period of time, then the client cache lag setting forces the next `OCIStmtExecute()` call to make a database call to check for such invalidations.

The cached result sets relevant to database invalidations are immediately invalidated, and no subsequent `OCIStmtExecute()` calls can match such result sets. The OCI statement handles currently fetching from these cached result sets, at the time such invalidations are received, can continue fetching from this (invalidated) cached result set.
The next OCIStmtExecute() call by the process may cache the new result set if there is space available in the cache. The OCI client result cache periodically reclaims unused memory.

If a session has a transaction open, OCI ensures that its queries that reference database objects changed in this transaction go to the server instead of the client cache.

This consistency mechanism ensures that the OCI cache is always close to committed database changes. If the OCI application has relatively frequent calls involving database round-trips due to queries that cannot be cached, (such as DMLs, OCILob calls, and so on) then these calls transparently keep the client cache up-to-date with database changes.

Note that sometimes when a table is modified, a trigger can cause another table to be modified. OCI client result cache is sensitive to all such changes.

When the session state is altered, for example, if NLS session parameters are modified, this can cause the query results to be different. The OCI result cache is sensitive to such changes and on subsequent query executions, returns the correct query result set. The current cached result sets are kept (and not invalidated) for any other session in the process to match; otherwise, these result sets get "Ruled" after a while. There are new result sets cached corresponding to the new session state.

If the application must keep track of all such database and session changes it can be cumbersome and prone to errors. Hence, OCI result cache transparently keeps the result sets consistent with any database or session changes.

The OCI client result cache does not require thread support in the client.

**Deployment Time Settings for Client Result Cache**

The client result cache has server initialization parameters and client configuration parameters for its deployment time settings.

These are the server initialization parameters:

- **CLIENT_RESULT_CACHE_SIZE**

  The default value is zero, implying that the client cache feature is disabled. To enable the client result cache feature, set the size to 32768 bytes (32 Kilobytes (KB)) or greater. This is the minimum size of the client per-process result set cache. All OCI client processes get this minimum size. This can be overridden by the sqlnet.ora configuration parameter OCI_RESULT_CACHE_MAX_SIZE only if this feature is enabled on the server by the CLIENT_RESULT_CACHE_SIZE initialization parameter.

  You can view the current default maximum size by displaying the value of the CLIENT_RESULT_CACHE_SIZE parameter. To increase this maximum size, you can set CLIENT_RESULT_CACHE_SIZE. However, because CLIENT_RESULT_CACHE_SIZE is a static parameter, you must include the `SCOPE = SPFILE` clause if you use an `ALTER SYSTEM` statement, and you must restart the database before any changes to this parameter take effect.

  Note that if the client result cache feature is disabled at the server, the client configuration parameter OCI_RESULT_CACHE_MAX_SIZE is ignored and the client result cache cannot be enabled at the client.

  The cache size can be set to the minimum of:

  (available client memory) and

  ((the possible number of result sets to be cached)
* (the average size of a row in a result set)
* (the average number of rows in a result set)).

---

**Note:** The client result cache has a maximum value of 2 GB; setting it higher causes a truncation to 2 GB.

Do not set the `CLIENT_RESULT_CACHE_SIZE` parameter during database creation, because that can cause errors.

---

- **CLIENT_RESULT_CACHE_LAG**

  The `CLIENT_RESULT_CACHE_LAG` initialization parameter enables you to specify the maximum amount of time in milliseconds that the client result cache can lag behind any changes in the database that affect its result sets. The default is 3000 milliseconds.

  You can view the current lag by displaying the value of the `CLIENT_RESULT_CACHE_LAG` parameter. To change this value, you can set `CLIENT_RESULT_CACHE_LAG`. However, because `CLIENT_RESULT_CACHE_LAG` is a static parameter, you must include the `SCOPE = SPFILE` clause if you use an `ALTER SYSTEM` statement, and you must restart the database before any changes to this parameter take effect.

- **Table annotation. Optional.** One can annotate read-only, read-mostly tables for result caching during deployment. No application-level changes are required.
  Note SQL result cache hints, if specified, override the table annotations. See Oracle Database SQL Language Reference for more information.

- **compatible**

  Specifies the release with which Oracle Database must maintain compatibility. This parameter must be set to 11.0.0.0 or higher for the client result cache to be enabled. If you want client caching on views, `compatible` must be set to 11.2.0.0 or higher.

**Client Configuration File**

A client configuration file is optional and overrides the cache parameters set in the server `init.ora` initialization file. These parameters are part of a `sqlnet.ora` file. The following optional parameters are available for client configuration:

- **OCI_RESULT_CACHE_MAX_SIZE** (optional) - Maximum size in bytes for the per-process query cache. Specifying a size less than 32768 in the client `sqlnet.ora` file disables the client result cache feature for client processes reading this `sqlnet.ora` file.

- **OCI_RESULT_CACHE_MAX_RSET_SIZE** (optional) - Maximum size of any result set in bytes in the per-process query cache.

- **OCI_RESULT_CACHE_MAX_RSET_ROWS** (optional) - Maximum size of any result set in rows in the per-process query cache.

Note that the cache lag cannot be set on the client.

**Client Cache Statistics**

On existing round-trips from the OCI client, OCI periodically sends statistics related to its client cache to the server. These statistics are stored in the `CLIENT_RESULT_CACHE_STATS$` view. Information such as the number of result sets successfully cached, number of cache hits, and number of cached result sets invalidated are stored here.
The number of cache misses for queries is at least equal to the number of Create Counts in client result cache statistics. More precisely, the cache miss count equals the number of server executions as seen in server Automatic Workload Repository (AWR) reports.

**See Also:**
- *Oracle Database Reference* for information about the `CLIENT_RESULT_CACHE_STAT$` view
- *Oracle Database Performance Tuning Guide* to find the client process IDs and session IDs for the sessions doing client caching

**Validation of the Client Result Cache**

The following sections provide some more information about performing validations of the client result cache.

**Timing Measurement**

First, to determine the performance gain of adding result cache hints to the queries, measure the time taken to run the queries without the `/*+ result_cache */` hints. Then add the `/*+ result_cache */` hints to the query and measure the time again. The difference in time is your performance gain.

**Using v$mystat**

Query the `v$mystat` view. To query this view, you must be granted permissions. Perform these two queries:

Query-1: Measures the “SQL*Net round-trips to and from the client” from `v$mystat`.

Query-2: Measures the “SQL*Net round-trips to and from the client” without the SQL result cache hint.

The difference between Query-2 and Query-1 queries is the number of round-trips that it usually takes without enabling client result cache.

Note that the Query-1 query itself would make some round-trips (approximately 2) in this calculation.

If you add a result cache hint to the query or add the `FORCE` table annotation to the query for table `emp` and perform the query again, the difference between Query-2 and Query-1 is much less.

**Using v$sqlarea**

Query the `v$sqlarea` view. To query this view, you must be granted permissions.

Run the following SQL statement:

```
SELECT COUNT(*) FROM emp
```

Reexecute this preceding SQL statement a few times.

Then query select executions, fetches, parse_calls from `v$sqlarea` where `sql_text` like '% from emp';

Next, add the result cache table hint for `emp` to the query.

Reexecute the query a few times.

With client caching, the values for column1, column2 are less.
Note that the preceding validations can also be performed with result cache table annotations.

**OCI Client-Side Result Cache and Server Result Cache**

The client-side result cache is a separate feature from the server result cache. The client-side result cache caches results of top-level SQL queries in OCI client memory, whereas the server result cache caches result sets in server SGA memory.

The server result cache may also cache query fragments. The client-side result caching can be enabled independently of the server result cache, though they both share the result cache SQL hints, table annotation, and session parameter `RESULT_CACHE_MODE`. See *Oracle Database Concepts* for more information about SQL query result cache. *Table 10–3* shows the specific result cache association for client-site result cache or server result cache, or both, with regard to setting specific parameters, running particular PL/SQL packages, and querying specific Oracle database views.

### Table 10–3 Setting Client-Side Result Cache and Server Result Cache

<table>
<thead>
<tr>
<th>Parameters, PL/SQL Package, and Database Views</th>
<th>Result Cache Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>client_result_cache_* parameters</td>
<td>client result cache</td>
</tr>
<tr>
<td><code>client_result_cache_size</code>,</td>
<td></td>
</tr>
<tr>
<td><code>client_result_cache_lag</code></td>
<td></td>
</tr>
<tr>
<td>SQL hints /*+ result_cache */,</td>
<td>client result cache,</td>
</tr>
<tr>
<td>/*+ no_result_cache */</td>
<td>server result cache</td>
</tr>
<tr>
<td><code>sqlnet.ora OCI_RESULT_CACHE*</code> parameters:</td>
<td>client result cache</td>
</tr>
<tr>
<td><code>OCI_RESULT_CACHE_MAX_SIZE</code></td>
<td></td>
</tr>
<tr>
<td><code>OCI_RESULT_CACHE_MAX_RSET_SIZE</code></td>
<td></td>
</tr>
<tr>
<td><code>OCI_RESULT_CACHE_MAX_RSET_ROWS</code></td>
<td></td>
</tr>
<tr>
<td>Statistics view: <code>client_result_cache_stats$</code></td>
<td>client result cache</td>
</tr>
<tr>
<td><code>result_cache_mode</code> parameter</td>
<td>client result cache,</td>
</tr>
<tr>
<td></td>
<td>server result cache</td>
</tr>
<tr>
<td>All other result_cache* parameters, for example, <code>result_cache_max_size</code></td>
<td>server result cache</td>
</tr>
<tr>
<td><code>Package DBMS_RESULT_CACHE</code></td>
<td></td>
</tr>
<tr>
<td>Statistics views <code>v$result_cache_</code>, <code>gv$result_cache_*</code>.</td>
<td>server result cache</td>
</tr>
<tr>
<td>For example, <code>v$result_cache_statistics</code>, <code>gv$result_cache_memory</code></td>
<td></td>
</tr>
<tr>
<td>Create table annotation</td>
<td>client result cache,</td>
</tr>
<tr>
<td></td>
<td>server result cache</td>
</tr>
<tr>
<td>Alter table annotation</td>
<td>client result cache,</td>
</tr>
<tr>
<td></td>
<td>server result cache</td>
</tr>
</tbody>
</table>

### Client Result Cache Demo Files

See the files `cdemoqc.sql`, `cdemoqc.c`, and `cdemoqc2.c` (all are in the demo directory for your operating system) for demonstration files for this feature.
Compatibility with Previous Releases

To use client result cache, applications must be relinked with Oracle Database Release 11.1 or later client libraries and be connected to an Oracle Database Release 11.1 or later database server. This feature is available to all OCI applications including JDBC Type II driver, OCCI, Pro*C/C++, and ODP.NET. The OCI drivers automatically pass the result cache hint to OCIStmtPrepare() and OCIStmtPrepare2() calls, thereby getting the benefits of caching.

Fault Diagnosability in OCI

Fault diagnosability was introduced into OCI in Oracle Database 11g 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. The diagnostic data is 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.

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.

ADR Base Location

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

1. OCI first looks in the file 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_11/oracle then the ADR base is /ade/chuck_11/oracle/log and the ADR home could be similar to /ade/chuck_11/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.

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

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: Oracle Database Net Services Reference

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.

See Also: Oracle Database Utilities for an introduction to ADRCI

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

See Also: "OCI Instant Client" on page 1-16
ADRCI: Release 11.1.0.5.0 - Beta on Wed May 2 15:53:06 2007

Copyright (c) 1982, 2007, Oracle. All rights reserved.

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 is "/ade/chuck_l3/oracle/log/oradiag_chuck"

Next, the SET BASE command is described:

adrci> help set base

Usage: SET BASE <base_str>

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.

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 .

adrci> quit

When ADRCI is started, the default ADR base is for the rdbms server. $ORACLE_HOME is set to /ade/chuck_l3/oracle:

% adrci

ADRCI: Release 11.1.0.5.0 - Beta on Wed May 2 16:16:55 2007

Copyright (c) 1982, 2007, Oracle. All rights reserved.

ADR base = "/ade/chuck_l3/oracle/log"

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

adrci> show base -product client
ADR base is "/ade/chuck_l3/oracle/log"
adrci> set base /ade/chuck_l3/oracle/log

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/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 rows fetched

adrci>

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

Controlling 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).
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()  
  - OCIIIntervalAdd()  
  - OCIIIntervalSubtract()  
  - OCIIIntervalFromTZ()  
  - 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

\(^1\) 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.
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
- Working with Objects in OCI
- Developing an OCI Object Application
- Type Inheritance
- Type Evolution

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

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

The OCI navigational calls are discussed in more detail beginning in "Developing an OCI Object Application" on page 11-5.

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.

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).

Basic Object Program Structure

The basic structure of an OCI application that uses objects is essentially the same as that for a relational OCI application, as described in "Overview of OCI Program Programming" on page 2-2. That paradigm is reproduced here, with extra information covering basic object functionality.

1. Initialize the OCI programming environment. You must initialize the environment in object mode.

   Your application 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), as described in Chapter 15.

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. "Fetching Embedded Objects" on page 11-11 describes fetching the entire 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:

- Chapter 2 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 Chapter 16
- "Representing Objects in C Applications” on page 11-5 for information about OTT and Chapter 15

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.

See Also: Oracle Database Object-Relational Developer’s Guide for more information about objects

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 REFs 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 11–1 and Example 11–2 show SQL examples that demonstrate the difference between these two types of persistent objects.

**Example 11–1 SQL Definition of Standalone Objects**

```sql
CREATE TYPE person_t AS OBJECT
  (name      varchar2(30),
   age       number(3));
CREATE TABLE person_tab OF person_t;
```

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.

**Example 11–2 SQL Definition of Embedded Objects**

```sql
CREATE TABLE department
  (deptno     number,
   deptname   varchar2(30),
   manager    person_t);
```

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.

**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: "Creating Objects" on page 11-23 for more information about using `OCIObjectNew()`

**Values**

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.
Developing an OCI Object Application

This section discusses the steps involved in developing a basic OCI object application. Each step discussed in "Basic Object Program Structure" on page 11-2 is described here in more detail.

Figure 11–1 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.

![Diagram of basic object operational flow]

Representing Objects in C Applications

Before an OCI application can work with object types, those types must exist in the database. Typically, you create types with SQL DDL statements, such as `CREATE TYPE`.

When the Oracle 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.

**Note:** Application programmers who want to use object representations other than the default structs generated by the object cache should see "Object Cache and Memory Management" on page 14-1.

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,
depthno NUMBER,
hiredate DATE,
salary NUMBER);

OTT produces the following C struct and corresponding NULL indicator struct:

```c
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, and are discussed in more detail in Chapter 12.

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:**
- "NULL Indicator Structure" on page 11-21
- Chapter 15 for more information about OTT

### 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: Chapter 14 for a detailed explanation of the object cache

### Making Database Connections

Once the OCI environment has been properly initialized, the application can connect to a server. This is accomplished through the standard OCI connect calls described in "OCI Programming Steps" on page 2-12. 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.

### 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` 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. See "Fetching Embedded Objects" on page 11-11 for more information.

In the following example, the application declares a text block that stores a SQL statement designed to retrieve a `REF` to a single employee object from an object table of employees (`emp_tab`) in the database, when given a particular employee number that is passed as an input variable (`:emp_num`) at runtime:

```c
typedef char text[100];

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, as described in Chapter 2:

1. Prepare an application request, using `OCISQntPrepare()`.
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, like the one declared in "Representing Objects in C Applications" on page 11-5:

   ```c
   OCIRef *empl_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 `OCISQntExecute()`.
5. Fetch the resulting REF into empl_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:

- "OCI Programming Steps" on page 2-12 for general information about preparing and executing SQL statements
- "Advanced Bind Operations in OCI" on page 5-7 and "Advanced Define Operations in OCI" on page 5-15 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 Appendix B.

Pinning an Object

Upon completion of the fetch step, your application has a REF, or pointer, to an object. The actual object is not currently available to work with. Before you can manipulate an object, it must be pinned. Pinning an object loads the object instance into the object cache, and enables you to access and modify the instance's attributes and follow references from that object to other objects, if necessary. Your application also controls when modified objects are written back to the server.

Note: This section deals with a simple pin operation involving a single object at a time. For information about retrieving multiple objects through complex object retrieval, see "Complex Object Retrieval" on page 11-15.

An application pins an object by calling the function OCIObjectPin(). The parameters for this function allow you to specify the pin option, pin duration, and lock option for the object.

Example 11–3 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" on page 11-7.

**Example 11–3 Pinning an Object**

```c
if (OCIObjectPin(env, err, empl_ref, (OCIComplexObject *) 0,
             OCI_PIN_ANY,
             OCI_DURATION_TRANS,
             OCI_LOCK_X, &empl) != OCI_SUCCESS)
    process_error(err);
```

In this example, process_error() represents an error-handling function. If the call to OCIObjectPin() returns anything but OCI_SUCCESS, the error-handling function is called. The parameters of the OCIObjectPin() function are as follows:

- env is the OCI environment handle.
- err is the OCI error handle.
- empl_ref is the reference that was retrieved through SQL.
- (OCIComplexObject *) 0 indicates that this pin operation is not utilizing complex object retrieval.
Developing an OCI Object Application

- **OCI_PIN_ANY** is the pin option. See "Pinning an Object Copy" on page 14-5 for more information.
- **OCI_DURATION_TRANS** is the pin duration. See "Object Duration" on page 14-11 for more information.
- **OCI_LOCK_X** is the lock option. See "Locking Objects for Update" on page 14-10 for more information.
- **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.

**See Also:**  "Simple Object Navigation" on page 14-14 for an example of navigation from one instance to another

**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.

**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. See "Marking Objects and Flushing Changes" on page 11-10 for more information.

For example, assume that the employee object in the previous section was pinned so that the employee's salary could be increased. Assume also that at this company, yearly salary increases are prorated for employees who have been at the company for less than 180 days.

For this example, 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 11–4** 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 11–4  Manipulating Object Attributes in OCI**

```c
/* 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 */
```
Developing an OCI Object Application

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);

Example 11–4 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.

See Also: Chapter 12 for more information about the OCI data types and the data type mapping and manipulation functions

Marking Objects and Flushing Changes

In Example 11–4, 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 OCITransCommit() 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.
Fetching Embedded Objects

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" on page 11-7. 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.

```sql
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:

```sql
CREATE TABLE clients
    ( name          varchar2(40),
    addr          address);
```

Your OCI application could then issue the following SQL statement:

```sql
SELECT addr FROM clients
WHERE name='BEAR BYTE DATA MANAGEMENT'
```

This statement would return an embedded address object from the clients table. The application could then use the values in the attributes of this object for other processing.

Your application should prepare and process this statement in the same way that it would handle any relational SQL statement, as described in Chapter 2:

- 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 "Representing Objects in C Applications" on page 11-5.

```c
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().

See Also:
- "OCI Support for Transactions" on page 8-1 for more information about OCITransCommit()
- "Creating Objects" on page 11-23 for information about transient and persistent objects
- "Object Meta-Attributes" on page 11-12 for information about seeing and checking object meta-attributes, such as dirty
Following this operation, you can access the attributes of the instance, as described in "Manipulating Object Attributes" on page 11-9, or pass the instance as an input parameter for another SQL statement.

**Note:** Changes made to an embedded instance can be made persistent only by executing a SQL `UPDATE` statement.

**See Also:** "OCI Programming Steps" on page 2-12 for more information about preparing and executing SQL statements

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.

**Persistent Object Meta-Attributes**

Table 11–1 shows the meta-attributes for *standalone* persistent objects.

<table>
<thead>
<tr>
<th>Meta-Attributes</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>existent</td>
<td>Does the object exist?</td>
</tr>
<tr>
<td>nullity</td>
<td>Null information of the instance</td>
</tr>
<tr>
<td>locked</td>
<td>Has the object been locked?</td>
</tr>
<tr>
<td>dirty</td>
<td>Has the object been marked as <em>dirtied</em>?</td>
</tr>
<tr>
<td>pinned</td>
<td>Is the object pinned?</td>
</tr>
<tr>
<td>allocation duration</td>
<td>See &quot;Object Duration&quot; on page 14-11.</td>
</tr>
<tr>
<td>pin duration</td>
<td>See &quot;Object Duration&quot; on page 14-11.</td>
</tr>
</tbody>
</table>

**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:

```c
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.

See Also: "OCIObjectGetProperty()" on page 18-23

**OCI_OBJECTPROP_LIFETIME**
This identifies whether the given object is a persistent object or a transient object or a value instance. The property argument must be a pointer to a variable of type OCIObjectLifetime. Possible values include:

- OCI_OBJECT_PERSISTENT
- OCI_OBJECT_TRANSIENT
- OCI_OBJECT_VALUE

**OCI_OBJECTPROP_SCHEMA**
This returns the schema name of the table in which the object exists. An error is returned if the given object points to a transient instance or a value. If the input buffer is not big enough to hold the schema name, an error is returned; the error message 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

See Also: "Object Duration" on page 14-11 for more information about durations

**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.

**Additional Attribute Functions**
OCI also provides functions that allow an application to set or check some of these attributes directly or indirectly, as shown in Table 11–2.

### Table 11–2  Set and Check Functions

<table>
<thead>
<tr>
<th>Meta-Attribute</th>
<th>Set with</th>
<th>Check with</th>
</tr>
</thead>
<tbody>
<tr>
<td>nullity</td>
<td>&lt;none&gt;</td>
<td>OCIObjectGetInd()</td>
</tr>
<tr>
<td>existence</td>
<td>&lt;none&gt;</td>
<td>OCIObjectExists()</td>
</tr>
<tr>
<td>locked</td>
<td>OCIObjectLock()</td>
<td>OCIObjectIsLocked()</td>
</tr>
<tr>
<td>dirty</td>
<td>OCIObjectMarkUpdate()</td>
<td>OCIObjectIsDirty()</td>
</tr>
</tbody>
</table>

**Transient Object Meta-Attributes**
Transient objects have no persistent attributes. Table 11–3 shows the following transient attributes.

### Table 11–3  Transient Meta-Attributes

<table>
<thead>
<tr>
<th>Transient Meta-Attributes</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>existent</td>
<td>Does the object exist?</td>
</tr>
<tr>
<td>pinned</td>
<td>Is the object being accessed by the application?</td>
</tr>
<tr>
<td>dirty</td>
<td>Has the object been marked as dirtied?</td>
</tr>
<tr>
<td>nullity</td>
<td>Null information of the instance.</td>
</tr>
<tr>
<td>allocation duration</td>
<td>See &quot;Object Duration&quot; on page 14-11.</td>
</tr>
</tbody>
</table>
Complex Object Retrieval

In Example 11–3 and Example 11–4, 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 REFS (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.

**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 SELECT privilege is needed for all prefetched objects. Objects in the complex object for which the application does not have SELECT privilege are not prefetched.

Implementing Complex Object Retrieval in OCI

Complex object retrieval (COR) allows an application to prefetch a complex object while fetching the root object. The complex object specifications are passed to the same OCIObjectPin() function used for simple objects.

An application specifies the parameters for complex object retrieval using a complex object retrieval handle. This handle is of type OCIComplexObject and is allocated in the same way as other OCI handles.

The complex object retrieval handle contains a list of complex object retrieval descriptors. The descriptors are of type OCIComplexObjectComp, and are allocated in the same way as other OCI descriptors.

Each COR descriptor contains a type REF and a depth level. The type REF specifies a type of reference to be followed while constructing the complex object. The depth level indicates how far a particular type of reference should be followed. Specify an integer value, or 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 REFs are followed to the depth specified in the COR handle. The COR handle can also be used to specify whether a collection attribute should be fetched separately on demand (out-of-line) as opposed to the default case of fetching it along with the containing object (inline).

The application uses OCIAttrSet() to set the attributes of a COR handle. The attributes are:

OCI_ATTR_COMPLEXOBJECT_LEVEL - the depth level
OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE - fetch collection attribute in an object type out-of-line

The application allocates the COR descriptor using OCIDescriptorAlloc() and then can set the following attributes:

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE - the type REF
OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_Level - the depth level for references of the preceding type

Once these attributes are set, the application calls OCIParmSet() 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.
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.

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.

```c
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 REFS, and lock the entire complex object in another round-trip using the array interface (OCIObjectArrayPin()).

Example of COR

Example 11–5 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 11–5 Using Complex Object Retrieval in OCI**

```c
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 */
OCTHandleAlloc((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 */
OCIAtrrSet( (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 */
OCIAtrrSet( (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);
OCTHandleFree((void *) corhp, (ub4) OCI_HTYPE_COMPLEXOBJECT);

/* iterate and print line items for this purchase order */
OCIIterCreate(envhp, errhp, po->line_items, &itr);
```

OCI Object-Relational Programming 11-19
/* get first line item */
OCIterNext(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); // Display the line item

    /* get next line item */
    OCIterNext(envhp, errhp, itr, (void **)liref, (void **)0, &eoc);
}

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
eexecute 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
the 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.

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:

- "Freeing an Object Copy" on page 14-7 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
- "Marking Objects and Flushing Changes" on page 11-10 for information about explicitly flushing an object or the entire cache

NULL Indicator Structure

If a column in a row of a database table has no value, then that column is said to be NULL, or to contain a NULL. Two different types of NULLs can apply to objects:

- Any attribute of an object can have a NULL value. This indicates that the value of that attribute of the object is not known.
- An object instance may be atomically NULL, 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 11–6 shows the C representations of types with their corresponding NULL indicator structures.

**Example 11–6  C Representations of Types with Their Corresponding NULL Indicator Structures**

```c
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;
```
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:

```c
person_ind *my_person_ind
if( my_person_ind -> _atomic == OCI_IND_NULL)
   printf ("instance is atomically NULL\n");
else
   if( my_person_ind -> fname == OCI_IND_NULL)
      printf ("fname attribute is NULL\n");
```

In the preceding example, the value of the atomic NULL indicator, or one of the attribute NULL indicators, is compared to the predefined value OCI_IND_NULL to test if it is NULL. The following predefined values are available for such a comparison:

- OCI_IND_NOTNULL, indicating that the value is not NULL
- OCI_IND_NULL, indicating that the value is NULL
- OCI_IND_BADNULL indicates that an enclosing object (or parent object) is NULL. This is used by PL/SQL, and may also be referred to as an INVALID_NULL. For example, if a type instance is NULL, then its attributes are INVALID_NULLs.

Use the function "OCIObjectGetInd()" on page 18-33 to retrieve the NULL indicator structure of an object.

If you update an attribute in its C structure, you must also set the NULL indicator for that attribute:

```c
obj->attr1 = string1;
OCIObjectGetInd(envhp, errhp, obj, &ind);
ind->attr1 = OCI_IND_NOTNULL;
```

See Also: Chapter 15 for more information about OTT-generated NULL indicator structures

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.

**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 11–4, and the corresponding null indicators are set to NOT NULL.

**Table 11–4  Attribute Values for New Objects**

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>REF</td>
<td>If an object has a REF attribute, the user must set it to a valid REF before</td>
</tr>
<tr>
<td></td>
<td>flushing the object or an error is returned</td>
</tr>
<tr>
<td>DATE</td>
<td>The earliest possible date that Oracle Database allows, which is midnight,</td>
</tr>
<tr>
<td></td>
<td>01-JAN-4712 BCE (equivalent to Julian day 1)</td>
</tr>
<tr>
<td>ANSI DATE</td>
<td>The earliest possible date that Oracle Database allows, 01-JAN-4712 BCE</td>
</tr>
<tr>
<td></td>
<td>(equivalent to Julian day 1)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The earliest possible date and time that Oracle Database allows, which is</td>
</tr>
<tr>
<td></td>
<td>midnight, 01-JAN-4712 BCE (equivalent to Julian day 1)</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>The earliest possible date and time that Oracle Database allows, which is</td>
</tr>
<tr>
<td></td>
<td>midnight, 01-JAN-4712 BCE (equivalent to Julian day 1) at UTC (0:0) time zone</td>
</tr>
<tr>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>The earliest possible date and time that Oracle Database allows, which is</td>
</tr>
<tr>
<td></td>
<td>midnight, 01-JAN-4712 BCE (equivalent to Julian day 1) at UTC (0:0) time zone</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>INTERVAL '0-0' YEAR TO MONTH</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>INTERVAL '0 0:0:0' DAY TO SECOND</td>
</tr>
<tr>
<td>FLOAT</td>
<td>0</td>
</tr>
<tr>
<td>NUMBER</td>
<td>0</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>0</td>
</tr>
<tr>
<td>RAW</td>
<td>Raw data with length set to 0. Note: the default value for a RAW attribute is</td>
</tr>
<tr>
<td></td>
<td>the same as that for a NULL RAW attribute.</td>
</tr>
<tr>
<td>VARCHAR2, NVARCHAR2</td>
<td>OCIString with 0 length and first char set to NULL. The default value is the</td>
</tr>
<tr>
<td></td>
<td>same as that of a NULL string attribute.</td>
</tr>
<tr>
<td>CHAR, NCHAR</td>
<td>OCIString with 0 length and first char set to NULL. The default value is the</td>
</tr>
<tr>
<td></td>
<td>same as that of a null string attribute.</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>OCIString with 0 length and first char set to NULL. The default value is the</td>
</tr>
<tr>
<td></td>
<td>same as that of a null string attribute.</td>
</tr>
<tr>
<td>VARRAY</td>
<td>Collection with 0 elements</td>
</tr>
<tr>
<td>NESTED TABLE</td>
<td>Table with 0 elements</td>
</tr>
</tbody>
</table>
Developing an OCI Object Application

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().

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.

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 11–7 shows how this process might be implemented to create a new object for the `emp_view` object view in the `HR` schema.

Example 11–7 Creating a New Object for an Object View

```c
void object_view_new ()
{
    void     *table;
    OCIRRef   *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,TRUE,&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 */
```

**Error Handling in Object Applications**

Error handling in OCI applications is the same whether or not the application uses objects. For more information about function return codes and error messages, see “Error Handling in OCI” on page 2-20.
Type Inheritance

Type inheritance of objects has many similarities to inheritance in C++ and Java. You can create an object type as a subtype of an existing object type. The subtype is said to inherit all the attributes and methods (member functions and procedures) of the supertype, which is the original type. Only single inheritance is supported; an object cannot have more than one supertype. The subtype can add new attributes and methods to the ones it inherits. It can also override (redefine the implementation) of any of its inherited methods. A subtype is said to extend (that is, inherit from) its supertype.

See Also: Oracle Database Object-Relational Developer’s Guide for a more complete discussion

As an example, a type Person_t can have a subtype Student_t and a subtype Employee_t. In turn, Student_t can have its own subtype, PartTimeStudent_t. A type declaration must have the flag NOT FINAL so that it can have subtypes. The default is FINAL, which means that the type can have no subtypes.

All types discussed so far in this chapter are FINAL. All types in applications developed before 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:

```sql
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:

```sql
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:

```sql
CREATE TYPE Employee_t UNDER Person_t
(empid NUMBER,
 mgr VARCHAR2(30));
```

See “OTT Support for Type Inheritance” on page 15-13 for the resulting structs generated by OTT for this example.

This subtype Student_t can have its own subtype, such as PartTimeStudent_t:

```sql
CREATE TYPE PartTimeStudent_t UNDER Student_t
(numhours NUMBER);
```

Substitutability

The benefits of polymorphism derive partially from the property substitutability. Substitutability allows a value of some subtype to be used by code originally written for the supertype, without any specific knowledge of the subtype being needed in advance. The subtype value behaves to the surrounding code, just like a value of the supertype would, even if it perhaps uses different mechanisms within its specializations of methods.
Instance substitutability refers to the ability to use an object value of a subtype in a context declared in terms of a supertype. REF substitutability refers to the ability to use a REF to a subtype in a context declared in terms of a REF to a supertype.

REF type attributes are substitutable; that is, an attribute defined as REF T can hold a REF to an instance of T or any of its subtypes.

Object type attributes are substitutable; an attribute defined to be of (an object) type T can hold an instance of T or any of its subtypes.

Collection element types are substitutable; if 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:

```sql
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:

```sql
Book_t('My Oracle Experience',
       Employee_t(12345, 'Joe', 'SF', 1111, NULL))
```

### NOT INSTANTIABLE Types and Methods

A type can be declared to be NOT INSTANTIABLE, which means that there is no constructor (default or user-defined) for the type. Thus, it 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:

```sql
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:

```sql
CREATE TYPE T AS OBJECT
  ( x NUMBER,
    NOT INSTANTIABLE MEMBER FUNCTION func1() RETURN NUMBER
  ) NOT INSTANTIABLE NOT FINAL;
```

A subtype of a NOT INSTANTIABLE type can override any of the NOT INSTANTIABLE methods of the supertype and provide concrete implementations. If there are any NOT INSTANTIABLE methods remaining, the subtype must also necessarily be declared as NOT INSTANTIABLE.

A NOT INSTANTIABLE subtype can be defined under an instantiable supertype. Declaring a NOT INSTANTIABLE type to be FINAL is not useful and is not allowed.

### OCI Support for Type Inheritance

The following calls support type inheritance.
OCIDescribeAny()
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 6–7 and Table 6–9 for attributes that OCIDescribeAny() can use to describe existing schema and subschema objects

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: Chapter 12 for details of the bind and define processes

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.

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.

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.

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.
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.

OTT Support for Type Inheritance
The Object Type Translator (OTT) supports type inheritance of objects by declaring first the inherited attributes in an encapsulated struct called "_super", followed by the new declared attributes. This is done because C does not support type inheritance.

See Also: "OTT Support for Type Inheritance" on page 15-13 for an example and discussion

Type Evolution
Adding, dropping, and modifying type attributes are supported. This concept is known as type evolution. It is discussed in 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" on page 14-17 for a discussion of the effect of type evolution on the object cache
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; it 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
- Mapping Oracle Data Types to C
- 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)
- Multilevel Collection Types
- REF (OCIRef)
- Object Type Information Storage and Access
- AnyType, AnyData, and AnyDataSet Interfaces
- Binding Named Data Types
- Defining Named Data Types
- Binding and Defining Oracle C Data Types
- SQLT_NTY Bind and Define Examples

**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, and described in Chapter 14.

The OCI client must allocate a descriptor before performing a bind or define operation. OCICStmtExecute() and OCICStmtFetch2() 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 "Initializing the Environment and the Object Cache" on page 11-6.

See Also: Oracle Database Concepts for detailed information about object types, attributes, and collection data types

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.

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:

```sql
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:

```sql
CREATE TABLE employees
(name varchar2(30),
birthday date,
```

See Also: Oracle Database Concepts for detailed information about object types, attributes, and collection data types

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.
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.

See Also: Table 15–1 for the available Oracle types and their C mappings you can use as object attribute types

An additional C type, OCIInd, is used to represent null indicator information corresponding to attributes of object types.

See Also: Chapter 15 for more information and examples about using OTT

OCI Type Mapping Methodology

Oracle followed a distinct design philosophy when specifying the mappings of Oracle predefined types. The current system has the following benefits and advantages:

- The actual representation of 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.

Manipulating C Data Types with OCI

In an OCI application, the manipulation of data may be as simple as adding together two integer variables and storing the result in a third variable:

```c
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 15–1 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 12–1 shows the various
function prefixes, along with example function names and the data types on which the
functions operate.

Table 12–1  Function Prefix Examples

<table>
<thead>
<tr>
<th>Function Prefix</th>
<th>Example</th>
<th>Operates on</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIColl</td>
<td>OCICollGetElem()</td>
<td>OCIColl, OCIIter, OCIArray</td>
</tr>
<tr>
<td>OCIDate</td>
<td>OCIDateDaysBetween()</td>
<td>OCIDate</td>
</tr>
<tr>
<td>OCIDateTime</td>
<td>OCIDateTimeSubtract()</td>
<td>OCIDate, OCIDateTime</td>
</tr>
<tr>
<td>OCIInterval</td>
<td>OCIIntervalToText()</td>
<td>OCIInterval</td>
</tr>
<tr>
<td>OCIIter</td>
<td>OCIIterInit()</td>
<td>OCIIter</td>
</tr>
<tr>
<td>OCINumber</td>
<td>OCINumberAdd()</td>
<td>OCINumber</td>
</tr>
<tr>
<td>OCIRaw</td>
<td>OCIRawResize()</td>
<td>OCIRaw *</td>
</tr>
<tr>
<td>OCIRef</td>
<td>OCIRefAssign()</td>
<td>OCIRef *</td>
</tr>
<tr>
<td>OCIString</td>
<td>OCIStringSize()</td>
<td>OCIString *</td>
</tr>
<tr>
<td>OCITable</td>
<td>OCITableLast()</td>
<td>OCITable *</td>
</tr>
</tbody>
</table>

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.

Precision of Oracle Number Operations

Oracle numbers have a precision of 38 decimal digits. All Oracle number operations
are accurate to the full precision, with the following exceptions:

- Inverse trigonometric functions are accurate to 28 decimal digits.
- Other transcendental functions, including trigonometric functions, are accurate to
  approximately 37 decimal digits.
- Conversions to and from native floating-point types have the precision of the
  relevant floating-point type, not to exceed 38 decimal digits.
Date (OCIDate)

The Oracle date format is mapped in C by the OCIDate type, which is an opaque C struct. Elements of the struct represent the year, month, day, hour, minute, and second of the date. The specific elements can be set and retrieved using the appropriate OCI functions.

The OCIDate 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: Chapter 19 for the prototypes and descriptions of all the functions

Date Example

Example 12–1 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 as described in "OCI Environment Initialization" on page 2-13. See "Object Cache Operations" on page 14-4 for information about pinning.

Example 12–1  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 */
 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 12–2 in the bind or define call.

<table>
<thead>
<tr>
<th>OCI Data Type</th>
<th>Type of Data</th>
<th>External Typecode for Binding/Defining</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIDateTime</td>
<td>ANSI DATE</td>
<td>SQLT_DATE</td>
</tr>
<tr>
<td>OCIDateTime</td>
<td>TIMESTAMP</td>
<td>SQLT_TIMESTAMP</td>
</tr>
<tr>
<td>OCIDateTime</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>SQLT_TIMESTAMP_TZ</td>
</tr>
<tr>
<td>OCIDateTime</td>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>SQLT_TIMESTAMP_LTZ</td>
</tr>
<tr>
<td>OCIInterval</td>
<td>INTERVAL YEAR TO MONTH</td>
<td>SQLT_INTERVAL_YM</td>
</tr>
</tbody>
</table>
The OCI functions that operate on datetime and interval data are listed in Table 12–3 and Table 12–4. More detailed information about these functions can be found in "OCI Date, Datetime, and Interval Functions" on page 19-24.

In general, functions that operate on OCIDateTime data are also valid for OCIDate data.

### Datetime Functions

The following functions operate on OCIDateTime values. Some of these functions also perform arithmetic operations on datetime and interval values. Some functions may only work for certain datetime types. The possible types are:

- SQLT_DATE - DATE
- SQLT_TIMESTAMP - TIMESTAMP
- SQLT_TIMESTAMP_TZ - TIMESTAMP WITH TIME ZONE
- SQLT_TIMESTAMP_LTZ - TIMESTAMP WITH LOCAL TIME ZONE

See the individual function descriptions listed in Table 12–3 for more information about input types that are valid for a particular function.

### Table 12–3  Datetime Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIDateTimeAssign()&quot; on page 19-42</td>
<td>Performs datetime assignment</td>
</tr>
<tr>
<td>&quot;OCIDateTimeCheck()&quot; on page 19-43</td>
<td>Checks if the given date is valid</td>
</tr>
<tr>
<td>&quot;OCIDateTimeCompare()&quot; on page 19-45</td>
<td>Compares two datetime values</td>
</tr>
<tr>
<td>&quot;OCIDateTimeConstruct()&quot; on page 19-46</td>
<td>Constructs a datetime descriptor</td>
</tr>
<tr>
<td>&quot;OCIDateTimeConvert()&quot; on page 19-48</td>
<td>Converts one datetime type to another</td>
</tr>
<tr>
<td>&quot;OCIDateTimeFromArray()&quot; on page 19-49</td>
<td>Converts an array containing a date to an OCIDateTime descriptor</td>
</tr>
<tr>
<td>&quot;OCIDateTimeFromText()&quot; on page 19-50</td>
<td>Converts the given string to Oracle datetime type in the OCIDateTime descriptor, according to the specified format</td>
</tr>
<tr>
<td>&quot;OCIDateTimeGetDate()&quot; on page 19-52</td>
<td>Gets the date (year, month, day) portion of a datetime value</td>
</tr>
<tr>
<td>&quot;OCIDateTimeGetTime()&quot; on page 19-53</td>
<td>Gets the time (hour, minute, second, fractional second) from datetime value</td>
</tr>
<tr>
<td>&quot;OCIDateTimeGetTimeZoneName()&quot; on page 19-54</td>
<td>Gets the time zone name portion of a datetime value</td>
</tr>
<tr>
<td>&quot;OCIDateTimeGetTimeZoneOffset()&quot; on page 19-55</td>
<td>Gets the time zone (hour, minute) portion of a datetime value</td>
</tr>
<tr>
<td>&quot;OCIDateTimeIntervalAdd()&quot; on page 19-56</td>
<td>Adds an interval to a datetime to produce a resulting datetime</td>
</tr>
<tr>
<td>&quot;OCIDateTimeIntervalSub()&quot; on page 19-57</td>
<td>Subtracts an interval from a datetime and stores the result in a datetime</td>
</tr>
</tbody>
</table>
Datetime Example

The code fragment in Example 12–2 shows how to use an OCIDateTime data type to select data from a TIMESTAMP WITH LOCAL TIME ZONE column.

Example 12–2  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),
0, (void **)0);

status = 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);

Interval Functions

The functions listed in Table 12–4 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIDateTimeSubtract()&quot; on page 19-58</td>
<td>Takes two datetimes as input and stores their difference in an interval</td>
</tr>
<tr>
<td>&quot;OCIDateTimeSysTimeStamp()&quot; on page 19-59</td>
<td>Gets the system current date and time as a time stamp with time zone</td>
</tr>
<tr>
<td>&quot;OCIDateTimeToArray()&quot; on page 19-60</td>
<td>Converts an OCIDateTime descriptor to an array</td>
</tr>
<tr>
<td>&quot;OCIDateTimeToText()&quot; on page 19-61</td>
<td>Converts the given date to a string according to the specified format</td>
</tr>
<tr>
<td>&quot;OCIDateTimeZoneToZone()&quot; on page 19-65</td>
<td>Converts the date from one time zone to another time zone</td>
</tr>
</tbody>
</table>
Number (OCINumber)

- SQLT_INTERVAL_DS - interval day to second

See the individual function descriptions for more detailed information.

See Also: "OCI Date, Datetime, and Interval Functions" on page 19-24 for complete lists of the names and purposes and more detailed information about these functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIIntervalAdd()&quot; on page 19-67</td>
<td>Adds two intervals to produce a resulting interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalAssign()&quot; on page 19-68</td>
<td>Copies one interval to another</td>
</tr>
<tr>
<td>&quot;OCIIntervalCheck()&quot; on page 19-69</td>
<td>Checks the validity of an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalCompare()&quot; on page 19-71</td>
<td>Compares two intervals</td>
</tr>
<tr>
<td>&quot;OCIIntervalDivide()&quot; on page 19-72</td>
<td>Divides an interval by an Oracle NUMBER to produce an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalFromNumber()&quot; on page 19-73</td>
<td>Converts an Oracle NUMBER to an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalFromText()&quot; on page 19-74</td>
<td>When given an interval string, converts the interval represented by the string</td>
</tr>
<tr>
<td>&quot;OCIIntervalFromTZ()&quot; on page 19-75</td>
<td>Returns an interval when given an input string of time zone form</td>
</tr>
<tr>
<td>&quot;OCIIntervalGetDaySecond()&quot; on page 19-76</td>
<td>Gets values of day, hour, minute, and second from an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalGetYearMonth()&quot; on page 19-77</td>
<td>Gets year and month from an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalMultiply()&quot; on page 19-78</td>
<td>Multiplies an interval by an Oracle NUMBER to produce an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalSetDaySecond()&quot; on page 19-79</td>
<td>Sets day, hour, minute, and second in an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalSetYearMonth()&quot; on page 19-80</td>
<td>Sets year and month in an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalSubtract()&quot; on page 19-81</td>
<td>Subtracts two intervals and stores the result in an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalToNumber()&quot; on page 19-82</td>
<td>Converts an interval to an Oracle NUMBER</td>
</tr>
<tr>
<td>&quot;OCIIntervalToText()&quot; on page 19-83</td>
<td>When given an interval, produces a string representing the interval</td>
</tr>
</tbody>
</table>

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.

See Also: Table 19-11 for the prototypes and descriptions for all the OCI NUMBER functions
OCINumber Examples

The code fragment in Example 12–3 shows how to manipulate an attribute of type OCINumber. The code fragment in Example 12–4 shows how to convert values in OCINumber format returned from OCIDescribeAny() calls to unsigned integers.

Example 12–3 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 */
Example 12–4 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 12–4  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);
    if (isError (status, _errhp))
        return 0;
    //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),
Fixed or Variable-Length String (OCIString)

Fixed or variable-length string data is represented to C programs as an `OCIString *`. The length of the string does not include the NULL character.

For binding and defining variables of type `OCIString *` use the external typecode `SQLT_VST`.

See Also: Table 19–16 for the prototypes and descriptions for all the string functions

String Functions

Table 12–5 shows the functions that allow the C programmer to manipulate an instance of a string.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIStringAllocSize()&quot; on page 19-150</td>
<td>Get allocated size of string memory in code points (Unicode) or bytes</td>
</tr>
<tr>
<td>&quot;OCIStringAssign()&quot; on page 19-151</td>
<td>Assign one string to another string</td>
</tr>
<tr>
<td>&quot;OCIStringAssignText()&quot; on page 19-152</td>
<td>Assign the source text string to the target string</td>
</tr>
<tr>
<td>&quot;OCIStringPtr()&quot; on page 19-153</td>
<td>Get a pointer to the text of a given string</td>
</tr>
<tr>
<td>&quot;OCIStringResize()&quot; on page 19-154</td>
<td>Resize the memory of a given string</td>
</tr>
<tr>
<td>&quot;OCIStringSize()&quot; on page 19-155</td>
<td>Get the size of a given string</td>
</tr>
</tbody>
</table>

String Example

Example 12–5 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 12–5  Manipulating an Attribute of Type OCIString

```c
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",
```
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.

See Also: Table 19–14 for the prototypes and descriptions for all the raw functions

Raw Functions

Table 12–6 shows the functions that perform OCIRaw operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIRawAllocSize()&quot;</td>
<td>Get the allocated size of raw memory in bytes</td>
</tr>
<tr>
<td>&quot;OCIRawAssignBytes()&quot;</td>
<td>Assign raw data (ub1 *) to OCIRaw *</td>
</tr>
<tr>
<td>&quot;OCIRawAssignRaw()&quot;</td>
<td>Assign one OCIRaw * to another</td>
</tr>
<tr>
<td>&quot;OCIRawPtr()&quot;</td>
<td>Get pointer to raw data</td>
</tr>
<tr>
<td>&quot;OCIRawResize()&quot;</td>
<td>Resize memory of variable-length raw data</td>
</tr>
<tr>
<td>&quot;OCIRawSize()&quot;</td>
<td>Get size of raw data</td>
</tr>
</tbody>
</table>

Raw Example

Example 12–6 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 12–6  Manipulating an Attribute of Type OCIRaw

```c
(ub4)strlen("This is a longer string.");
/* 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);
```
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.

See Also: "Nested Table Manipulation Functions" on page 12-16

You can allocate an instance of a varray or nested table using OCIObjectNew() and free it using OCIObjectFree().

See Also: "OCI Collection and Iterator Functions" on page 19-3 for the prototypes and descriptions for these functions

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().

See Also: "Nested Table Manipulation Functions" on page 12-16

Note: Indexes passed to collection functions are zero-based.

Collection Data Manipulation Functions

Table 12–7 shows the generic functions that manipulate collection data.

<table>
<thead>
<tr>
<th>Table 12–7 Collection Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
</tr>
<tr>
<td>&quot;OCICollAppend()&quot; on page 19-4</td>
</tr>
<tr>
<td>&quot;OCICollAssign()&quot; on page 19-5</td>
</tr>
<tr>
<td>&quot;OCICollAssignElem()&quot; on page 19-6</td>
</tr>
<tr>
<td>&quot;OCICollGetElem()&quot; on page 19-7</td>
</tr>
</tbody>
</table>
Collection Scanning Functions

Table 12–8 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 12–8 Collection Scanning Functions](image)

Varray/Collection Iterator Example

Example 12–7 creates and uses a collection iterator to scan through a varray.

Example 12–7 Using Collection Data Manipulation Functions

```c
OCIEnv *envhp;
OCIError *errhp;
text *text_ptr;
sword status;
OCIArray *clients;
OCIString *client_elem;
OCIIter *iterator;
boolean eoc;
void *elem;
OCIIInd *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:
");
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);

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 12–9 shows the functions that are specific
to nested tables. They should not be used on varrays.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCITableDelete()&quot; on page 19-157</td>
<td>Delete an element at a given index</td>
</tr>
<tr>
<td>&quot;OCITableExists()&quot; on page 19-158</td>
<td>Test whether an element exists at a given index</td>
</tr>
<tr>
<td>&quot;OCITableFirst()&quot; on page 19-159</td>
<td>Return the index for the first existing element of a table</td>
</tr>
<tr>
<td>&quot;OCITableLast()&quot; on page 19-160</td>
<td>Return the index for the last existing element of a table</td>
</tr>
</tbody>
</table>
Multilevel Collection Types

Object-Relational Data Types in OCI

Nested Table Element Ordering
When a nested table is fetched into the object cache, its elements are given a transient ordering, numbered from zero to the number of elements, minus 1. For example, a table with 40 elements would be numbered from 0 to 39.

You can use these position ordinals to fetch and assign the values of elements (for example, fetch to element \( i \), or assign to element \( j \), where \( i \) and \( j \) are valid position ordinals for the given table).

When the table is copied back to the database, its transient ordering is lost. Delete operations may be performed against elements of the table. Delete operations create transient holes; that is, they do not change the position ordinals of the remaining table elements.

Nested Table Locators
You can retrieve a locator to a nested table. A locator is like a handle to a collection value, and it contains information about the database snapshot 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 \texttt{RETURN AS LOCATOR} specification.

See Also: Oracle Database SQL Language Reference

You can use the \texttt{OCICollIsLocator()} function to determine whether a collection is locator-based or not.

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()

**Multilevel Collection Type Example**

The following types and tables are used for Example 12–8.

```
type_1 (a NUMBER, b NUMBER)
NT1 TABLE OF type_1
NT2 TABLE OF NT1
```

The code fragment in Example 12–8 iterates over the multilevel collection.

**Example 12–8 Using Multilevel Collection Data Manipulation Functions**

```c
...
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));
...
```

**REF (OCIRef)**

A REF (reference) is an identifier to an object. It is an opaque structure that uniquely locates the object. An object may point to another object by way of a REF.
In C applications, the REF is represented by OCIRef*.

See Also: Table 19–15 for the prototypes and descriptions for all the REF manipulation functions

REF Manipulation Functions

Table 12–10 shows the functions that perform REF operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIRefAssign()&quot;</td>
<td>Assign one REF to another</td>
</tr>
<tr>
<td>&quot;OCIRefClear()&quot;</td>
<td>Clear or nullify a REF</td>
</tr>
<tr>
<td>&quot;OCIRefFromHex()&quot;</td>
<td>Convert a hexadecimal string to a REF</td>
</tr>
<tr>
<td>&quot;OCIRefHexSize()&quot;</td>
<td>Return the size of a hexadecimal string representation of REF</td>
</tr>
<tr>
<td>&quot;OCIRefsEqual()&quot;</td>
<td>Compare two REFS for equality</td>
</tr>
<tr>
<td>&quot;OCIRefsNull()&quot;</td>
<td>Test whether a REF is NULL</td>
</tr>
<tr>
<td>&quot;OCIRefToHex()&quot;</td>
<td>Convert a REF to a hexadecimal string</td>
</tr>
</tbody>
</table>

REF Example

Example 12–9 tests two REFS for NULL, compares them for equality, and assigns one REF to another. Note the double indirection in the call to OCIRefAssign().

Example 12–9 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 = OCIRefsEqual(envhp, ref1, ref2);
printf("After first OCIRefsEqual:\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 = OCIRefsEqual(envhp, ref1, ref2);
printf("After second OCIRefsEqual:\n");
if(refs_equal)
  printf("REFS equal\n");
else
  printf("REFS not equal\n");
Object Type Information Storage and Access

The OCI data types and type descriptors are discussed in this section.

Descriptor Objects

When a given type is created with the CREATE TYPE statement, it is stored in the server and associated with a type descriptor object (TDO). In addition, the database stores descriptor objects for each data attribute of the type, each method of the type, each parameter of each method, and the results returned by methods. Table 12–11 lists the OCI data types associated with each type of descriptor object.

<table>
<thead>
<tr>
<th>Information Type</th>
<th>OCI Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>OCIType</td>
</tr>
<tr>
<td>Type Attributes</td>
<td>OCITypeElem</td>
</tr>
<tr>
<td>Collection Elements</td>
<td></td>
</tr>
<tr>
<td>Method Parameters</td>
<td></td>
</tr>
<tr>
<td>Method Results</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>OCITypeMethod</td>
</tr>
</tbody>
</table>

Several OCI functions (including OCIBindObject() and OCIObjectNew()) require a TDO as an input parameter. An application can obtain the TDO by calling OCITypeByName(), which gets the type’s TDO in an OCIType variable. Once you obtain the TDO, you can pass it, as necessary, to other calls.

AnyType, AnyData, and AnyDataSet Interfaces

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.
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 12–10, use `OCITypeAddAttr()` to add the attribute information to the type. After adding information for the last attribute, you must call `OCITypeEndCreate()`.

**Example 12–10 Using Type Interfaces to Construct Object Types**

```c
OCITypeBeginCreate( ...) /* Begin Type Creation */
OCIDescriptorAlloc(...)
OCIAttrSet(...)
OCITypeAddAttr(...) /* Add attribute 1 */
OCIAttrSet(...)
OCITypeAddAttr(...) /* Add attribute 2 */
...
OCITypeEndCreate(...) /* End Type Creation */
```

For collection types, as shown in Example 12–11, use `OCITypeSetCollection()` to set the information on the collection element type. Subsequently, call `OCITypeEndCreate()` to finish construction.

**Example 12–11 Using Type Interfaces to Construct Collection Types**

```c
OCITypeBeginCreate( ...) /* Begin Type Creation */
OCIDescriptorAlloc(...)
OCIAttrSet(...)
OCITypeSetCollection(...) /* Set information on collection element */
OCITypeEndCreate(...) /* End Type Creation */
```

You can use the `OCIDescribeAny()` call to obtain the `OCIType` corresponding to a persistent type.

**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.

Obtaining the OCIType for Persistent Types
You can use the OCIDescribeAny() call to obtain the OCIType corresponding to a persistent type, as 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.

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.

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)
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).

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().
Any 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 12–12.

**Example 12–12 Using Special Construction and Access Calls for Improved Performance**

```c
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 */

or

OCIAnyDataCollAddElem(...) /* Element-wise construction for collections */

OCIAnyDataEndCreate(...) /* End OCIAnyData Creation */
```

**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_TYPECODEVARCHAR2 to the output values SQLT_VST (which corresponds to the OCIStrinct mapping) with a character set form of SQLCS_IMPLICIT. OCI_TYPECODE_NVARCHAR2 also returns SQLT_VST (OCIStrinct mapping is used by OCIAnyData API) with a character set form of SQLCS_NCHAR.

See Also: "OCIAnyDataTypeCodeToSqlt()" on page 21-29

**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:
Binding Named Data Types

This section provides information about binding named data types (such as objects and collections) and REFS.

Named Data Type Binds

For a named data type (object type or collection) bind, a second bind call is necessary following OCIBindByName() or OCIBindByPos(). 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(), OCIBindArrayOfStruct(), and OCIBindObject().

See Also:

■ "Fetching Embedded Objects" on page 11-11 for information about using these data types to fetch an embedded object from the database
■ "Information for Named Data Type and REF Binds" on page 12-26
■ "Descriptor Objects" on page 12-20

Binding REFS

As with named data types, binding REFS is a two-step process. First, call OCIBindByName() or OCIBindByPos(), and then call OCIBindObject().

REFS are bound using the SQLT_REF data type. When SQLT_REF is used, then the program variable being bound must be of type OCIRef *.

See Also: Chapter 21 for complete descriptions of all the calls in these interfaces
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" on page 11-7 for information about binding and pinning REFS to objects
- "Information for Named Data Type and REF Binds" on page 12-26 for additional important information

**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.

**Information Regarding Array Binds**

For doing array binds of named data types or REFS, 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" on page 3-16 for more information about SQLT_NTY

**Defining Named Data Types**

This section provides information about defining named data types (for example, objects, collections) and REFS.

**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(), specifying SQLT_NTY in the dty parameter. Following OCIDefineByPos(), the application must call OCIObject() to set up additional attributes pertaining to a named data type define. In this case, the data buffer pointer in OCIObject() 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" on page 12-27 for more important information about defining named data types

### Defining REF Output Variables

As with named data types, defining for a REF output variable is a two-step process. The first step is a call to `OCIDefineByPos()`, and the second is a call to `OCIDefineObject()`. Also as with named data types, the `SQLT_REF` 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 as described in "Working with Objects in OCI" on page 11-2.

**See Also:** "Information for Named Data Type and REF Defines, and PL/SQL OUT Binds" on page 12-27 for more important information about defining REFs

### 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. See "Binding Placeholders in PL/SQL" on page 5-4 for more information.

- 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).
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(). See Chapter 18 for details about OCIObjectNew() and OCIObjectFree().

Note: There is no change to the behavior of OCIDefineObject() when the user does not preallocate the object memory and instead initializes the output variable to null pointer value. In this case, the object 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.

Information About Array Defines
To perform array defines of named data types or REFS, 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.

Binding and Defining Oracle C Data Types
Previous chapters of this book have discussed OCI bind and define operations. "Binding Placeholders in OCI" on page 4-4 discussed the basics of OCI bind operations, whereas "Defining Output Variables in OCI" on page 4-12 discussed the basics of OCI define operations. Information specific to binding and defining named data types and REFS was described in Chapter 5.

The sections covering basic bind and define functionality showed how an application could use a scalar variable or array of scalars as an input (bind) value in a SQL statement, or as an output (define) buffer for a query.
The sections covering named data types and REFs showed how to bind or define an object or reference. "Pinning an Object" on page 11-8 expanded on this to talk about pinning object references, "Fetching Embedded Objects" on page 11-11 discussed fetching embedded instances, and "Object Navigation" on page 14-14 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 12–12 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>
<thead>
<tr>
<th>Data Type</th>
<th>C Mapping</th>
<th>OCI External Data Type and Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle NUMBER</td>
<td>OCINumber</td>
<td>VARNUM (SQLT_VNU)</td>
</tr>
<tr>
<td>Oracle DATE</td>
<td>OCIDate</td>
<td>SQLT_ODT</td>
</tr>
<tr>
<td>BLOB</td>
<td>OCIlobLocator *</td>
<td>SQLT_BLOB</td>
</tr>
<tr>
<td>CLOB, NCLOB</td>
<td>CIlobLocator *</td>
<td>SQLTY_LOB</td>
</tr>
<tr>
<td>VARCHAR2, NVARCHAR2</td>
<td>OCIString *</td>
<td>SQLT_VST 1</td>
</tr>
<tr>
<td>RAW</td>
<td>OCIRaw *</td>
<td>SQLT_LVB 1</td>
</tr>
<tr>
<td>CHAR, NCHAR</td>
<td>OCIString *</td>
<td>SQLT_VST</td>
</tr>
<tr>
<td>Object</td>
<td>struct *</td>
<td>Named Data Type (SQLT_NTY)</td>
</tr>
<tr>
<td>REF</td>
<td>OCIRef *</td>
<td>REF (SQLT_REF)</td>
</tr>
<tr>
<td>VARRAY</td>
<td>OCIArray *</td>
<td>Named Data Type (SQLT_NTY)</td>
</tr>
<tr>
<td>Nested Table</td>
<td>OCITable *</td>
<td>Named Data Type (SQLT_NTY)</td>
</tr>
<tr>
<td>DATETIME</td>
<td>OCIDateTime *</td>
<td>See &quot;Datetime and Interval (OCIDateTime, OCIInterval)&quot; on page 12-6.</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>OCIInterval *</td>
<td>See &quot;Datetime and Interval (OCIDateTime, OCIInterval)&quot; on page 12-6.</td>
</tr>
</tbody>
</table>

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 section presents examples of how to use C-mapped data types in an OCI application.

**See Also:** Chapter 3 for a discussion of OCI external data types, and a list of data typecodes

**Bind and Define Examples**

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:

```sql
CREATE TYPE person AS OBJECT
(name varchar2(30),
```
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;
```

See Also: Chapter 15 for a complete discussion of OTT

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" on page 5-15. 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:

```c
text *selstmt = (text *) "SELECT emp.salary FROM employees
                             WHERE emp.name = 'MONGO';"

OCINumber mongo_sal;
...
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.

### Salary Update Examples

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:

  ```sql
  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:

```c
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.

**Method 1 - Fetch, Convert, Assign**

**Example 12–13** 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 12–13   Method 1 for a Salary Update: Fetch, Convert, and Assign**

```c
#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 */
```

**Method 2 - Fetch and Assign**

This method (Example 12–14) 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 12–14   Method 2 for a Salary Update: Fetch and Assign, No Convert**

```c
#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 */
```

**Method 3 - Direct Fetch**

This method (Example 12–15) 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 12–15  Method 3 for a Salary Update: Direct Fetch**

```c
#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 */
```

**Summary and Notes**

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.

### 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.

**SQLT_NTY Bind Example**

**Example 12–16** shows how to use the **SQLT_NTY** named data type in the bind call including **OCIBindObject()**.

```c
/*
 ** This example performs a SQL insert statement
 /*
```
**SQLT_NTY Bind and Define Examples**

Example 12–17 shows how to use the SQLT_NTY named data type in the define call including OCIDefineObject().

**Example 12–17 Using the SQLT_NTY Define Call Including OCIDefineObject()**

/*

*/
** This example executes a SELECT statement from a table that includes
** an object.
*/

tvoid 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, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
        (OCISnapshot *) NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

    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));
The direct path loading functions are used to load data from external files into tables and partitions.

This chapter contains these topics:
- Direct Path Loading Overview
- Direct Path Loading of Object Types
- Direct Path Loading in Pieces
- Direct Path Context Handles and Attributes for Object Types

Direct Path Loading Overview

The direct path load interface enables an OCI application to access the direct path load engine of 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 13–1 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.
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

See Also: "Direct Path Context Handle (OCIDirPathCtx) Attributes" on page A-63

Data Types Supported for Direct Path Loading

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:

- "Accessing Column Parameter Attributes" on page A-71 for information on setting or retrieving the data type of a column
- Table 3–2 for information about data types
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" on page A-62 and all the descriptions of direct path attributes that follow

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 13–1.

**Example 13–1   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);
```

OCI Direct Path Function Context

**See Also:** Oracle Database Object-Relational Developer’s Guide for more information about the data types supported

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 13–2.

**Example 13–2 Passing the Handle Type to Allocate the Function Context**

```c
OCIDirPathCtx *dpctx; /* direct path context */
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
sword error;

test = OCIHandleAlloc((void *)dpctx, (void **)&dpfnctx,
                     OCI_HTYPE_DIRPATH_FN_CTX,
                     (size_t)0, (void **)0);
```

Note that the parent handle of a direct path function context is always the direct path context handle. A direct path function context handle is freed with OCIHandleFree():

test = OCIHandleFree(dpfnctx, OCI_HTYPE_DIRPATH_FN_CTX);

**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 13–3 shows explicit allocation of the direct path column array handle.

**Example 13–3 Explicit Allocation of Direct Path Column Array Handle**

```c
OCIDirPathCtx *dpctx;      /* direct path context */
OCIDirPathColArray *dpca;  /* direct path column array */
sword error;

test = OCIHandleAlloc((void *)dpctx, (void **)&dpca,
                     OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                     (size_t)0, (void **)0);
```

A direct path column array handle is freed with OCIHandleFree():

test = OCIHandleFree(dpca, OCI_HTYPE_DIRPATH_COLUMN_ARRAY);

Example 13–4 shows that a direct path function column array handle is allocated in almost the same way.

**Example 13–4 Explicit Allocation of Direct Path Function Column Array Handle**

```c
OCIDirPathFuncCtx *dpfnctx;   /* direct path function context */
OCIDirPathColArray *dpfnc;    /* direct path function column array */
sword error;

test = OCIHandleAlloc((void *)dpfnctx, (void **)&dpfnc,
                     OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
                     (size_t)0, (void **)0);
```
A direct path function column array is freed with `OCIHandleFree()`:

```c
error = OCIHandleFree(dpfnca, OCI_HTYPE_DIRPATH_FN_COL_ARRAY);
```

Freeing an `OCIDirPathColArray` handle also frees the column array associated with the handle.

**Direct Path Stream**

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)`. A direct path stream is a linear representation of Oracle table data. The conversion operations always append to the end of the stream. Load operations always start from the beginning of the stream. After a stream is completely loaded, the stream must be reset by calling `OCIDirPathStreamReset()`.

Example 13–5 shows a direct path stream handle allocated with `OCIHandleAlloc()`. The parent handle is always an `OCIDirPathCtx` handle.

**Example 13–5  Allocating a Direct Path Stream Handle**

```c
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);
```

A direct path stream handle is freed using `OCIHandleFree()`.

```c
error = OCIHandleFree(dpstr, OCI_HTYPE_DIRPATH_STREAM);
```

Freeing an `OCIDirPathStream` handle also frees the stream buffer associated with the handle.

**Direct Path Interface Functions**

The functions listed in this section are used with the direct path load interface.

**See Also:** "Direct Path Loading Functions" on page 17-106 for detailed descriptions of each function

Operations on the direct path context are performed by the functions in Table 13–1.

**Table 13–1  Direct Path Context Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OCIDirPathAbort()</code></td>
<td>Terminates a direct path operation</td>
</tr>
<tr>
<td><code>OCIDirPathDataSave()</code></td>
<td>Executes a data savepoint</td>
</tr>
<tr>
<td><code>OCIDirPathFinish()</code></td>
<td>Commits the loaded data</td>
</tr>
<tr>
<td><code>OCIDirPathFlushRow()</code></td>
<td>Flushes a partially loaded row from the server. This function is deprecated.</td>
</tr>
<tr>
<td><code>OCIDirPathLoadStream()</code></td>
<td>Loads the data that has been converted to direct path stream format</td>
</tr>
</tbody>
</table>
Operations on the direct path column array are performed by the functions in Table 13–2.

**Table 13–2  Direct Path Column Array Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIDirPathColArrayEntryGet()</td>
<td>Gets a specified entry in a column array</td>
</tr>
<tr>
<td>OCIDirPathColArrayEntrySet()</td>
<td>Sets a specified entry in a column array to a specific value</td>
</tr>
<tr>
<td>OCIDirPathColArrayRowGet()</td>
<td>Gets the base row pointers for a specified row number</td>
</tr>
<tr>
<td>OCIDirPathColArrayReset()</td>
<td>Resets the row array state</td>
</tr>
<tr>
<td>OCIDirPathColArrayToStream()</td>
<td>Converts from a column array format to a direct path stream format</td>
</tr>
</tbody>
</table>

Operations on the direct path stream are performed by the function OCIDirPathStreamReset() that resets the direct stream state.

**Limitations and Restrictions of the Direct Path Load Interface**

The direct path load interface has the following limitations that are the same as SQL*Loader:

- Triggers are not supported.
- Referential integrity constraints are not supported.
- Clustered tables are not supported.
- Loading of remote objects is not supported.
- LONGs must be specified last.
- SQL strings that return LOBs, objects, or collections are not supported.
- Loading of VARRAY columns is not supported.
- All partitioning columns must come before any LOBs. This is because you must determine what partition the LOB goes into before you start writing to it.

**Direct Path Load Examples for Scalar Columns**

This section describes the direct path load examples for scalar columns.

**Data Structures Used in Direct Path Loading Example**

Example 13–6 shows the data structure used in Example 13–7 through Example 13–17.

**Example 13–6  Data Structures Used in Direct Path Loading Examples**

```c
/* load control structure */
struct loadctl
{
    ub4 nrow_ctl;    /* number of rows in column array */
```
Example 13–7 shows the header file cdemodp.h from the demo directory, which defines several structs.

### Example 13–7 Contents of the Header File cdemodp.h

```c
#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 */
};
```
Outline of an Example of a Direct Path Load for Scalar Columns

Example 13–8 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.
Example 13–8 Use of OCI Direct Path Interfaces

```c
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->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
        OCIAttrSet((void  *)dpctx, (ub4)OCI_HTYPE_DIRPATH_CTX, 
            (ub4)OCI_ATTR_SUB_NAME, ctlp->errhp_ctl));
    OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
        OCIAttrSet((void  *)dpctx, (ub4)OCI_HTYPE_DIRPATH_CTX, 
            (ub4)OCI_ATTR_SCHEMA_NAME, ctlp->errhp_ctl));

    Additional attributes, such as OCI_ATTR_SUB_NAME and OCI_ATTR_SCHEMA_NAME, are also 
    set here. After the attributes have been set, prepare the load.

    OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, 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 13–9. Also note that errors are returned with the 
environment handle associated with the direct path context.

Example 13–9 Allocating the Column Array and Stream Handles

```
Get the Number of Rows and Columns
Get the number of rows and columns in the column array just allocated, as shown in Example 13–10.

Example 13–10  Getting the Number of Rows and Columns

```c
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
   OCIAttrGet(ctlp->dpca_ctl, (ub4)OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
     &ctlp->nrow_ctl, 0, OCI_ATTR_NUM_ROWS,
     ctlp->errhp_ctl));

OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
   OCIAttrGet(ctlp->dpca_ctl, (ub4)OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
     &ctlp->ncol_ctl, 0, OCI_ATTR_NUM_COLS,
     ctlp->errhp_ctl));
```

Set the Input Data Fields
Set the input data fields to their corresponding data columns, as shown in Example 13–11.

Example 13–11  Setting Input Data Fields

```c
ub4            rowoff;                          /* column array row offset */
ub4            clen;                                      /* column length */
ub1            cflg;                                  /* column state flag */
ub1           *cval;                             /* column character value */

OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
   OCIDirPathColArrayEntrySet(ctlp->dpca_ctl, ctlp->errhp_ctl,
     rowoff, colp->id_col,
     cval, clen, cflg));
```

Reset 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 13–12.

Example 13–12  Resetting the Column Array State

```c
(void) OCIDirPathColArrayReset(ctlp->dpca_ctl, ctlp->errhpCtl);
```

Reset the Stream State
Reset the stream state to start a new stream, as shown in Example 13–13. Otherwise, data in the stream is appended to existing data.

Example 13–13  Resetting the Stream State

```c
(void) OCIDirPathStreamReset(ctlp->dpstr_ctl, ctlp->errhp_ctl);
```

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 13–14.
Example 13–14  Converting Data to Stream Format

```c
ub4    rowcnt;       /* number of rows in column array */
ub4    startoff;    /* starting row offset into column array */

/* convert array to stream, filter out bad records */
ocierr = OCIDirPathColArrayToStream(ctlp->dpca_ctl, ctlp->dpctx_ctl,
                                     ctlp->dpstr_ctl, ctlp->errhp_ctl,
                                     rowcnt, startoff);
```

Load the Stream

Note that the position in the stream is maintained internally to the stream handle, along with offset information for the column array that produced the stream. When the conversion to stream format is done, the data is appended to the stream, as shown in Example 13–15. 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.

Example 13–15  Loading the Stream

```c
/* load the stream */
ociret = OCIDirPathLoadStream(ctlp->dpctx_ctl, ctlp->dpstr_ctl,
                              ctlp->errhp_ctl);
```

Finish the Direct Path Load

Finish the direct path load as shown in Example 13–16.

Example 13–16  Finishing the Direct Path Load Operation

```c
/* finish the direct path load operation */
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIDirPathFinish(ctlp->dpctx_ctl, ctlp->errhp_ctl));
```

Free the Direct Path Handles

Free all the direct path handles allocated, as shown in Example 13–17. Note that the direct path column array and stream handles are freed before the parent direct path context handle is freed.

Example 13–17  Freeing the Direct Path Handles

```c
/* free up server data structures for the load */
ociret = OCIHHandleFree((void  *)ctlp->dpca_ctl,
                        OCI_HTYPE_DIRPATH_COLUMN_ARRAY);
ociret = OCIHHandleFree((void  *)ctlp->dpstr_ctl,
                        OCI_HTYPE_DIRPATH_STREAM);
ociret = OCIHHandleFree((void  *)ctlp->dpctx_ctl,
                        OCI_HTYPE_DIRPATH_CTX);
```

Using a Date Cache in Direct Path Loading of Dates in OCI

The date cache feature provides improved performance when loading Oracle date and 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**

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 OCI_DIRPATHPrepare() 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.

**OCI_ATTR_DIRPATH_DCACHE_NUM**

This attribute is used to query the current number of entries in a date cache.

**OCI_ATTR_DIRPATH_DCACHE_MISSES**

This attribute is used to query the current number of date cache misses. If 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.
Direct Path Loading of Object Types

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.

OCI_ATTR_DIRPATH_DCACHE_DISABLE
Setting this attribute to 1 indicates that the date cache should be disabled if the size is exceeded. Note that this attribute cannot be changed or set after OCIDirPathPrepare() has been called.

The default (= 0) is to not disable a cache on overflow. When not disabled, the cache is searched to avoid conversions, but overflow input date value entries 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" on page A-63

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

See Also: Table B–1 for a listing of the programs demonstrating direct path loading that are available with your Oracle Database installation

Direct Path Loading of Nested Tables
Nested tables are stored in a separate table. Using the direct path loading API, a nested table is loaded separately from its parent table with a foreign key, called a SETID, to link the two tables together.

Note:
- Currently, the SETIDs must be user-supplied and are not system-generated.
- When loading the parent and child tables separately, it is possible 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.
Direct Path Loading of Object Types

Describing a Nested Table Column and Its 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:

```c
    ub1 flg = 1;
    sword error;

    error = OCIAttrSet((void *)colDesc,
                        OCI_DTYPE_PARAM,
                        (void *)&flg, (ub4)0,
                        OCI_ATTR_DIRPATH_SID, ctlp->errhp_ctl);
```

Direct Path Loading of Column Objects

A column object is a table column that is defined as an object. Currently only the default constructor, which consists of all of the constituent attributes, is supported.

Describing a Column Object

To describe a column object and its object attributes, use a direct path function context. Describing a column object requires setting its object constructor. Describing object attributes is similar to describing a list of scalar columns.

To describe a column object:

**Note:**
- Nested column objects are supported.
- The steps 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:

   ```c
   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:

   ```c
   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:

   ```c
   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.
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 13–18 shows how to allocate a child column array for a column object.

Example 13–18 Allocating a Child Column Array for a Column Object

```c
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);
```

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()`.

**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.

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 LONGs.

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.

Describing a SQL String Column

```
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,
```

Note: Steps that are different from loading scalar data are in italic.
Direct Path Loading of Object Types

5. Set the column’s SQL string in OCI_ATTR_NAME in the function context.

```c
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.

```c
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:
   a. Get the column descriptor for the object attribute with OCI_DTYPE_PARAM.
   b. The order in which the SQL string arguments are defined does not matter. The order does not have to match the order used in the SQL string.
   c. Set the attribute’s column name with OCI_ATTR_NAME.
   d. Use the naming convention for SQL string arguments.
   e. The argument names must match the bind variable names used in the SQL string in content but not in case. For example, if the SQL string is “substr(:INPUT_STRING, 3, 5)”, then it is acceptable if you give the argument name as “input_string”.
   f. If an argument is used multiple times in a SQL string, declaring it once and counting it as one argument only is correct.
   g. Set the external column type (the type of the data that is to be passed to the direct path API) with OCI_ATTR_DATA_TYPE.
   h. Set any other external column attributes (maximum data size, precision, scale, and so on).
   i. Free the handle to the column descriptor.

10. Set the function context OCIDirPathFuncCtx that was created in Step 4 into the parent column object’s parameter handle with OCI_ATTR_DIRPATH_FN_CTX.

Allocating the Column Array for SQL String Columns
When 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 13–19 shows how to allocate a child column array for a SQL string column.

Example 13–19 Allocating a Child Column Array for a SQL String Column

```c
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);
```

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:
   
   a. For each of its function argument columns:
      
      Set the data in the child column array using `OCIDirPathColArrayEntrySet()`.
   
   b. Set the SQL string column’s data into the column array by passing the address of its
      child column array handle to `OCIDirPathColArrayEntrySet()`.

2. Else if the column is NULL:

   Set the SQL string column data into the column array by passing a NULL address
   for the data, a length of 0, and an `OCI_DIRPATH_COL_NULL` flag to
   `OCIDirPathColArrayEntrySet()`.

This process is similar to that for column objects.

**See Also:** "OCI_DIRPATH_COL_ERROR" on page 13-17 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.

Direct Path Loading of REF Columns

The REF type is a pointer, or reference, to a row object in an object table.

Describing the REF Column

Describing the arguments to a REF column is similar to describing the list of columns
to be loaded for a table.
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.

```c
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
sword error;
```

```c
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.

```c
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
text *ref_tbl;     /* column’s reference table */
sword error;
```

```c
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.
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`.

**Allocating the Column Array for a REF Column**

Example 13–20 shows how to allocate a child column array for a REF column.

**Example 13–20  Allocating a Child Column Array for a REF Column**

```c
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);
```

**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:**
   
a. **For each of its REF argument columns:**
      
      Set its data in the child column array using `OCIDirPathColArrayEntrySet()`.
   
b. **Set the REF column’s data into the column array by passing the address of its child column array handle to `OCIDirPathColArrayEntrySet()`**.

2. **Else if the column is NULL:**
   
   Set the REF column’s data into the column array by passing a NULL address for the data, a length of 0, and an `OCI_DIRPATH_COL_NULL` flag to `OCIDirPathColArrayEntrySet()`.
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.

Inheritance Hierarchy

Figure 13–2 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.

`Person` (Name, Address)

`Student` (Units, GPA)  `Employee` (Manager, Deptid)

`ParttimeEmployee` (Hours)

Recall that for an object type to be inheritable, the object type definition must specify that it is inheritable. 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. See Oracle Database Object-Relational Developer’s Guide for more information about defining `FINAL` and `NOT FINAL` types.

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: "OCI_DIRPATH_COL_ERROR" on page 13-17
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.

To describe a NOT FINAL column of type X (where X is an object or a REF), see "Direct Path Loading of Column Objects" on page 13-15 or "Direct Path Loading of REF Columns" on page 13-20. 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.

Allocating the Column Array
Allocating the column array is the same as for a FINAL column of the same type.

Loading the Data into the Column Array
Loading the data into the column array is the same as for a FINAL column of the same type.

Direct Path Loading of Object Tables
An object table is a table in which each row is an object (or row object). Each column in the table is an object attribute.

Describing an Object Table
Describing an object table is very similar to describing a non-object table. Each object attribute is a column in the table. The only difference is that you may need to describe the OID, which could be system-generated, user-generated, or primary-key-based.

To describe an object table:

Note: Steps that are different from loading a non-object table are in 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 13–21 shows that allocating the column array for the object table is the same as allocating a column array for a non-object table.

Example 13–21 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);

Loading Data into the Column Array
Loading data into the column array is the same as loading data into a non-object table.

Direct Path Loading a NOT FINAL Object Table

A NOT FINAL object table supports inheritance and a FINAL object table cannot.

Describing a NOT FINAL Object Table
Describing a NOT FINAL object table of a fixed derived type is very similar to describing a FINAL object table.

To describe a NOT FINAL object table of a fixed derived type:

Note: Steps that are different from 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.
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 LONGs 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.

See Also: “OCI_DIRPATH_COL_ERROR” on page 13-17 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.

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.

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" on page 13-17 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.

Direct Path Context Handles and Attributes for Object Types

The following discussion gives the supplemental details of the handles and attributes 
that are listed in Appendix A.

Direct Path Context Attributes

There is one direct path context attribute.

OCI_ATTR_DIRPATH_OBJ_CONSTR
Indicates the object type to load into a NOT FINAL object table.

```c

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);
```

Direct Path Function Context and Attributes

Here is a summary of the attributes for function context handles.

See Also: "Direct Path Context Handle (OCIDirPathCtx) Attributes" on 
page A-63

OCI_ATTR_DIRPATH_OBJ_CONSTR
Indicates the object type to load into a substitutable object table.

```c

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);
```

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.

**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 13–22 shows the pseudocode that illustrates the preceding rules and values.

### Example 13–22 Specifying Values for the OCI_ATTR_DIRPATH_EXPR_TYPE Attribute

```c
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,
  OCI_ATTR_DIRPATH_EXPR_TYPE, ctlp->errhp_ctl);
```

### 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. And, 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" on page A-65

### 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.

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.
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 13–23 shows that when you set a function context as a column attribute, OCI_ATTR_DIRPATH_FN_CTX is used in the OCIAttrSet() call.

Example 13–23 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, ctp->errhp_ctl);

Attributes for column parameter context handles follow.

See Also: “Direct Path Column Parameter Attributes” on page A-71

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 \texttt{OCI\_ATTR\_DIRPATH\_OID}.

- **SQL string argument**
  1. Set the attribute’s column name with \texttt{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 \texttt{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 \texttt{OCI\_ATTR\_NAME}.
     The order of the \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{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.

\textbf{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.
**OCI_ATTR_DIRPATH_SID**
Indicates that a column is a nested table's SETID column. Required if loading to a nested table.

```c
ub1 flg = 1;
sword error;

error = OCIAttrSet((void *)&colDesc, OCI_DTYPE_PARAM, (void *)&flg, (ub4)0, OCI_ATTR_DIRPATH_SID, ctlp->errhp_ctl);
```

**OCI_ATTR_DIRPATH_OID**
Indicates that a column is an object table's object ID column.

```c
ub1 flg = 1;
sword error;

error = OCIAttrSet((void *)&colDesc, OCI_DTYPE_PARAM, (void *)&flg, (ub4)0, OCI_ATTR_DIRPATH_OID, ctlp->errhp_ctl);
```

---

**Direct Path Function Column Array Handle for Nonscalar Columns**

**See Also:** "Direct Path Function Column Array Handle (OCIDirPathColArray) Attributes" on page A-69

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 13–24** shows how to allocate a child column array for a function context.

**Example 13–24  Allocating a Child Column Array for a Function Context**

```c
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);
```

**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.
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
- OCI Support for XML

### 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.

See Also:  "Pinning an Object Copy" on page 14-5

The object cache is created when the OCI environment is initialized using OCIEnvCreate() with mode set to OCI_OBJECT.

The object cache maintains a fast lookup table for mapping REFs 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 14–1.
Figure 14–1  Object Cache

The object cache tracks the objects that are currently in memory, maintains references to the objects, manages automatic object swapping, and tracks object meta-attributes.

Cache Consistency and Coherency

The object cache does not automatically maintain value coherency or consistency between object copies and their corresponding objects in the database. In other words, if an application makes changes to an object copy, the changes are not automatically applied to the corresponding object in the database, and vice versa. The cache provides operations such as flushing a modified object copy to the database and refreshing a stale object copy with the latest value from the database to enable the program to maintain some coherency.

Note: Oracle 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.
Object Cache Parameters

The object cache has two important parameters associated with it, which are attributes of the environment handle:

- **OCI_ATTR_CACHE_MAX_SIZE** – The maximum cache size
- **OCI_ATTR_CACHE_OPT_SIZE** – The optimal cache size

These parameters refer to levels of cache memory usage, and they help 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:

\[
\text{maximum_cache_size} = \text{optimal_size} + \text{optimal_size} \times \frac{\text{max_size_percentage}}{100}
\]

Next, represent the algorithm in terms of environment handle attributes.

\[
\text{maximum_cache_size} = \text{OCI_ATTR_CACHE_OPT_SIZE} + \text{OCI_ATTR_CACHE_OPT_SIZE} \times \frac{\text{OCI_ATTR_CACHE_MAX_SIZE}}{100}
\]

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:** See "Environment Handle Attributes" on page A-2 for more information

Object Cache Operations

This section describes the most important functions that the object cache provides to operate on object copies.

**See Also:** "OCI Navigational Functions" on page 14-15 for a list of all the OCI navigational, cache, and object management functions

Pinning and Unpinning

Pinning an object copy enables the application to access it in the cache by dereferencing the **REF** to it.

Unpinning an object indicates to the cache that the object currently is not being used. Objects should be unpinned when they are no longer needed to make them eligible for implicit freeing by the cache, thus freeing up memory.

Freeing

Freeing an object copy removes it from the cache and frees its memory.
Marking and Unmarking
Marking an object notifies the cache that an object copy has been updated in the cache and the corresponding object must be updated in the server when the object copy is flushed.

Unmarking an object removes the indication that the object has been updated.

Flushing
Flushing an object writes local changes made to marked object copies in the cache to the corresponding objects in the server. When this happens, the copies in the object cache are unmarked.

Refreshing
Refreshing an object copy in the cache replaces it with the latest value of the corresponding object in the server.

Note 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:

```c
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.

Loading and Removing Object Copies
Pin, unpin, and free functions are discussed in this section.

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 \textit{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 \textit{recent} (\texttt{OCI\_PIN\_RECENT}) option is specified, there are two possibilities:
  - If in the same transaction the object copy has been previously pinned using the \textit{latest} or \textit{recent} option, the \textit{recent} option becomes equivalent to the \textit{any} option.
  - If the previous condition does not apply, the \textit{recent} option becomes equivalent to the \textit{latest} option.

When the program pins an object, the program also specifies one of two possible values for the pin duration: \textit{session} or \textit{transaction}. The data type of the duration is \texttt{OCIDuration}.

- If the pin duration is \textit{session} (\texttt{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 \texttt{OCIObjectUnpin()}).

- If the pin duration is \textit{transaction} (\texttt{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:

\begin{verbatim}
SELECT VALUE(t) FROM t WHERE REF(t) = :r
\end{verbatim}

In this statement, \(t\) is the object table storing the object, \(r\) is the \texttt{REF}, and the fetched value becomes the value of the object copy in the cache.

Because the object cache effectively executes a separate \texttt{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 \textit{recent} or \textit{latest} are read-consistent with each other because the \texttt{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).
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.

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 OCICacheFree() 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.

See Also: “Object Cache Parameters” on page 14-4 for more information

For memory management reasons, it is important that applications unpin objects when they are no longer needed. This makes these objects available for aging out of the cache, and makes it easier for the cache to free memory when necessary.

OCI does not provide a function to free unreferenced objects in the client-side cache.

Making Changes to Object Copies

Functions for marking and unmarking object copies are discussed in this section.

Marking an Object Copy

An object copy can be created, updated, and deleted locally in the cache. If the object copy is created in the cache (by calling OCIObjectNew()), the object copy is marked for...
insert by the object cache, so that the object 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.

**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.

**Synchronizing Object Copies with the Server**

Cache and server synchronization operations (flushing, refreshing) are discussed in this section.

**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() on page 18-7.

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 OCICacheFlush() 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 OCICacheFlush(), the objects are flushed in the order that they are marked dirty. The performance of this flush operation can be considerably improved by setting the OCI_ATTR_CACHE_ARRAYFLUSH attribute in the environment handle.

See Also: "Environment Handle Attributes" on page A-2

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.

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 14–1 after being refreshed.

<table>
<thead>
<tr>
<th>Object Attribute</th>
<th>Status After Refresh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existent</td>
<td>Set to appropriate value</td>
</tr>
<tr>
<td>Pinned</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Flushed</td>
<td>Reset</td>
</tr>
<tr>
<td>Allocation duration</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Pin duration</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

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.
Object Locking

OCI functions related to object locking are discussed in this section.

Lock Options

When pinning an object, you can specify whether the object should be locked or not through lock options. When an object is locked, a server-side lock is acquired, 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.

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:

```sql
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.

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.
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.

Commit and Rollback in Object Applications

When a transaction is committed (OCITransCommit()), all marked objects are flushed to the server. If an object copy is pinned with a transaction duration, the object copy is unpinned.

When a transaction is rolled back, all marked objects are unmarked. If an object copy is pinned with a transaction duration, the object copy is unpinned.

Object Duration

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.

### Durations Example

Table 14–2 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>
<thead>
<tr>
<th>Time</th>
<th>Application Action</th>
<th>Function</th>
<th>Object 1</th>
<th>Object 2</th>
<th>Object 3</th>
<th>Object 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Establish connection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T2</td>
<td>Create object 1 - allocation</td>
<td>OCIObjectNew()</td>
<td>Exists</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>duration = connection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>Start Transaction1</td>
<td>OCITransStart()</td>
<td>Exists</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T6</td>
<td>SQL - fetch REF to object 2</td>
<td>-</td>
<td>Exists</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T7</td>
<td>Pin object 2 - pin duration =</td>
<td>OCIObjectPin()</td>
<td>Exists</td>
<td>Pinned</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>transaction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T8</td>
<td>Process application data</td>
<td>-</td>
<td>Exists</td>
<td>Pinned</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T9</td>
<td>Commit Transaction1</td>
<td>OCITransCommit()</td>
<td>Exists</td>
<td>Unpinned</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T10</td>
<td>Start Transaction2</td>
<td>OCITransStart()</td>
<td>Exists</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T11</td>
<td>Create object 3 - allocation</td>
<td>OCIObjectNew()</td>
<td>Exists</td>
<td>-</td>
<td>Exists</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>duration = transaction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T12</td>
<td>SQL - fetch REF to object 4</td>
<td>-</td>
<td>Exists</td>
<td>-</td>
<td>Exists</td>
<td>-</td>
</tr>
<tr>
<td>T13</td>
<td>Pin object 4 - pin duration =</td>
<td>OCIObjectPin()</td>
<td>Exists</td>
<td>-</td>
<td>Exists</td>
<td>Pinned</td>
</tr>
<tr>
<td></td>
<td>connection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T14</td>
<td>Commit Transaction2</td>
<td>OCITransCommit()</td>
<td>Exists</td>
<td>-</td>
<td>Deleted</td>
<td>Pinned</td>
</tr>
<tr>
<td>T15</td>
<td>Terminate session1</td>
<td>OCIDurationEnd()</td>
<td>Exists</td>
<td>-</td>
<td>-</td>
<td>Pinned</td>
</tr>
</tbody>
</table>
Object Cache and Memory Management

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 14–1.

Example 14–1 Object Type Representation of a Department Row

```sql
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 */
```

The C representation of the DEPARTMENT is shown in Example 14–2.

Example 14–2 C Representation of a Department Row

```c
struct department
{
  OCISTring * dep_name;
  OCINumber budget;
  struct person manager;
  OCIArrray * employees;
};
typedef struct department department;
```

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 14–2, 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.

---

**Table 14–2 (Cont.) Example of Allocation and Pin Durations**

<table>
<thead>
<tr>
<th>Time</th>
<th>Application Action</th>
<th>Function</th>
<th>Object 1</th>
<th>Object 2</th>
<th>Object 3</th>
<th>Object 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T16</td>
<td>Start Transaction3</td>
<td>OCITransStart()</td>
<td>Exists</td>
<td>-</td>
<td>-</td>
<td>Pinned</td>
</tr>
<tr>
<td>T17</td>
<td>Process application data</td>
<td>-</td>
<td>Exists</td>
<td>-</td>
<td>-</td>
<td>Pinned</td>
</tr>
<tr>
<td>T18</td>
<td>Commit Transaction3</td>
<td>OCITransCommit()</td>
<td>Exists</td>
<td>-</td>
<td>-</td>
<td>Pinned</td>
</tr>
<tr>
<td>T19</td>
<td>Terminate connection</td>
<td>-</td>
<td>Deleted</td>
<td>-</td>
<td>-</td>
<td>Unpinned</td>
</tr>
</tbody>
</table>

See Also:

- The descriptions of OCIObjectNew() and OCIObjectPin() in Chapter 18 for specific information about parameter values that can be passed to these functions.
- "Creating Objects" on page 11-23 for information about freeing up an object's memory before its allocation duration has expired.
Object Navigation

This section discusses how OCI applications can navigate through graphs of objects in the object cache.

Simple Object Navigation

In Example 14–1 and Example 14–2, the object retrieved by the application was a simple object, whose attributes were all scalar values. If an application retrieves an object with an attribute that is a REF to another object, the application can use OCI calls to traverse the object graph and access the referenced instance.

As an example, consider the following declaration for a new type in the database:

```sql
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:

```sql
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 14–2 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 14–2 Object Graph of person_t Instances](image-url)
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:

```c
OCIObjectPin(env, err, pers_1->mother, OCI_PIN_ANY, OCI_DURATION_TRANS, OCI_LOCK_X, (OCIComplexObject *) 0, &pers_2);
```

In this case, an OCI fetch operation is not required to retrieve the second instance. The application could then pin the father instance of `person1`, or it could operate on the reference links of `person2`.

---

**Note:** Attempting to pin a NULL or dangling REF results in an error on the `OCIObjectPin()` call.

---

**OCI Navigational Functions**

This section provides a brief summary of the available OCI navigational functions. The functions are grouped according to their general functionality.

**See Also:** Chapter 18 for more detailed descriptions of each of these 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.

**Pin/Unpin/Free Functions**

The functions in Table 14–3 are available to pin, unpin, or free objects.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OCICacheFree()</code></td>
<td>Free all instances in the cache</td>
</tr>
<tr>
<td><code>OCICacheUnpin()</code></td>
<td>Unpin persistent objects in cache or connection</td>
</tr>
<tr>
<td><code>OCIObjectArrayPin()</code></td>
<td>Pin an array of references</td>
</tr>
<tr>
<td><code>OCIObjectFree()</code></td>
<td>Free and unpin a standalone instance</td>
</tr>
<tr>
<td><code>OCIObjectPin()</code></td>
<td>Pin an object</td>
</tr>
<tr>
<td><code>OCIObjectPinCountReset()</code></td>
<td>Unpin an object to zero pin count</td>
</tr>
</tbody>
</table>
Flush and Refresh Functions

The functions in Table 14–4 are available to flush modified objects to the server.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCICacheFlush()</td>
<td>Flush modified persistent objects in cache to server</td>
</tr>
<tr>
<td>OCIObjectFlush()</td>
<td>Flush a modified persistent object to the server</td>
</tr>
<tr>
<td>OCICacheRefresh()</td>
<td>Refresh pinned persistent objects in the cache</td>
</tr>
<tr>
<td>OCIObjectRefresh()</td>
<td>Refresh a single persistent object</td>
</tr>
</tbody>
</table>

Mark and Unmark Functions

The functions in Table 14–5 allow an application to mark or unmark an object by modifying one of its meta-attributes.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIObjectMarkDeleteByRef()</td>
<td>Mark an object deleted when given a REF</td>
</tr>
<tr>
<td>OCIObjectMarkUpdate()</td>
<td>Mark an object as updated (dirty)</td>
</tr>
<tr>
<td>OCIObjectMarkDelete()</td>
<td>Mark an object deleted or delete a value instance</td>
</tr>
<tr>
<td>OCICacheUnmark()</td>
<td>Unmark all objects in the cache</td>
</tr>
<tr>
<td>OCIObjectUnmark()</td>
<td>Mark a given object as updated</td>
</tr>
<tr>
<td>OCIObjectUnmarkByRef()</td>
<td>Mark an object as updated, when given a REF</td>
</tr>
</tbody>
</table>

Object Meta-Attribute Accessor Functions

The functions in Table 14–6 allow an application to access the meta-attributes of an object.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIObjectExists()</td>
<td>Get existence status of an instance</td>
</tr>
<tr>
<td>OCIObjectFlushStatus()</td>
<td>Get the flush status of an instance</td>
</tr>
<tr>
<td>OCIObjectGetInd()</td>
<td>Get null structure of an instance</td>
</tr>
<tr>
<td>OCIObjectIsDirty()</td>
<td>Has an object been marked as updated?</td>
</tr>
<tr>
<td>OCIObjectIsLocked()</td>
<td>Is an object locked?</td>
</tr>
</tbody>
</table>

Other Functions

The functions in Table 14–7 provide additional object functionality for OCI applications.
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: "Type Evolution" on page 11-30

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.
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:

```c
xmlctx *OCIXmlDbInitXmlCtx (OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp,
          ocixmldbparam *params, ub4 num_params);

void OCIXmlDbFreeXmlCtx (xmlctx *xctx);
```

XML Data on the Server

XML data on the server can be operated on with OCI statement calls. You can bind and define XMLType values using xmlodcnodc, as with other object instances. OCI statements are used to select XML data from the server. This data can be used in the C DOM functions directly. Similarly, the values can be bound back to SQL statements directly.

Using OCI XML DB Functions

To initialize and terminate the XML context, use the functions OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx() respectively. The header file ocixmldb.h is used with the unified C API.

Example 14–3 is a code fragment of a tested example that shows how to perform operations with the C API.

**Example 14–3  Initializing and Terminating XML Context with a C API**

```c
#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;
    OCIError *errhp;
    OCIEnv *envhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCIStmt *stmthp;
    OCIServer *srvhp;
    OCIDuration dur;
    OCIStmt *stmthp;
    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 */
    ...
}

**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.

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).

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.

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).
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.

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.

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.

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
This chapter discusses the Object Type Translator (OTT), which is used to map database object types and named collection types to C structs for use in OCI applications.

This chapter contains these topics:

■ OTT Overview
■ What Is the Object Type Translator?
■ OTT Command Line
■ Intype File
■ OTT Data Type Mappings
■ Outtype File
■ Using OTT with OCI Applications
■ OTT Reference

OTT Overview

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.
What Is the Object Type Translator?

The Object Type Translator (OTT) converts database definitions of object types and named collection types into C struct declarations 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 15–1.

```
Example 15–1 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
};
```

Then the header file generated by OTT (demo.h) includes, among other items, the declarations shown in Example 15–2.

```
Example 15–2 Contents of the Generated Header File demo.h

struct emptype
{
    OCIString * name;
};
```
What Is the Object Type Translator?

Using the Object Type Translator with OCI

Example 15–3 shows what a sample implementation file (demov.c) produced by this command contains.

Example 15–3  Contents of the demov.c File

```c
#ifndef OCI_ORACLE
#include <oci.h>
#endif

sword demov(OCIEnv *env, OCIError *err)
{
    sword status = OCITypeVTInit(env, err);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
            'HR', 2,
            'EMPTYPE', 7,
            '$8.0', 4);
    return status;
}
```

Parameters in the intype file control the way generated structs are named. In this example, the struct name `emptype` matches the database type name `emptype`. The struct name is in 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.

See Also: "OTT Data Type Mappings" on page 15-8 for more information about these 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.

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.
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.

Command Line

Parameters (also called options) set on the command line override any set elsewhere.

See Also: "OTT Command Line" on page 15-4

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.

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" on page 15-6

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.

See Also: "Using the Object Type Translator for Windows" on page D-5
OTT Command-Line Invocation Example

Example 15–4 shows how to invoke OTT from the command line.

Example 15–4 Invoking OTT from the Command Line

ott userid=bren intype=demoin.typ outtype=demoout.typ code=c \ hfile=demo.h initfile=demov.c

Note: No spaces are permitted around the equal sign (=).

The following sections describe the elements of the command line used in this example.

See Also: "OTT Reference" on page 15-19 for a detailed discussion of the various OTT command-line options

OTT
Causes OTT to be invoked. It must be the first item on the command line.

USERID
Specifies the database connection information that OTT uses.
In Example 15–4, OTT attempts to connect with user name bren and is then prompted for the password.

INTYPE
Specifies the name of the intype file that is used.
In Example 15–4, the name of the intype file is specified as demoin.typ.

OUTTYPE
Specifies the name of the outtype file. When OTT generates the C header file, it also writes information about the translated types into the outtype file. This file contains an entry for each of the types that is translated, including its version string, and the header file to which its C representation was written.
In Example 15–4, 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.

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.

**HFILE**
Specifies the name of the C header file to which the generated structs should be
written.
In Example 15-4, 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 recompile.
---

**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 recompile.
---

**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 15-5 shows a simple user-created intype file.

**Example 15-5 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
```

Example 15-5 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.

See Also: "CASE" on page 15-24

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.

See Also: "Structure of the Intype File" on page 15-25 for a more detailed specification of the structure of the intype file and the available options

Using the Object Type Translator with OCI 15-7
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 15–6, and its corresponding OTT-generated struct declarations in Example 15–7.

Example 15–6 Object Type Definition for Employee

CREATE TYPE employee AS OBJECT
   (   name       VARCHAR2(30),
   empno      NUMBER,
   deptno     NUMBER,
   hiredate   DATE,
   salary$    NUMBER);

The OTT output, assuming CASE=LOWER and no explicit mappings of type or attribute names, is shown in Example 15–7.

Example 15–7 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;

See Also: "Null Indicator Structs" on page 15-12 for an explanation of the indicator struct (struct employee_ind)

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.

Mapping Object Data Types to C

This section describes the mappings of Oracle object attribute types to C types generated by OTT. The "OTT Type Mapping Example" on page 15-10 includes
examples of many of these different mappings. Table 15–1 lists the mappings from types that you can use as attributes to object data types that are generated by OTT.

Table 15–1 Object Data Type Mappings for Object Type Attributes

<table>
<thead>
<tr>
<th>Object Attribute Types</th>
<th>C Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFILE</td>
<td>OCIFileLocator*</td>
</tr>
<tr>
<td>BLOB</td>
<td>OCIFileLocator * or OCIFileLocator *</td>
</tr>
<tr>
<td>CHAR(N), CHARACTER(N), NCHAR(N)</td>
<td>OCIString *</td>
</tr>
<tr>
<td>CLOB, NCLOB</td>
<td>OCIFileLocator * or OCIFileLocator *</td>
</tr>
<tr>
<td>DATE</td>
<td>OCIDate</td>
</tr>
<tr>
<td>ANSI DATE</td>
<td>OCIDateTime *</td>
</tr>
<tr>
<td>TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>OCIDateTime *</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND</td>
<td>OCIDateTime</td>
</tr>
<tr>
<td>DEC, DEC(N), DEC(N,N)</td>
<td>OCINumber</td>
</tr>
<tr>
<td>DECIMAL, DECIMAL(N), DECIMAL(N,N)</td>
<td>OCINumber</td>
</tr>
<tr>
<td>FLOAT, FLOAT(N), DOUBLE PRECISION</td>
<td>OCINumber</td>
</tr>
<tr>
<td>BINARY_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>BINARY_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>INT, INTEGER, SMALLINT</td>
<td>OCINumber</td>
</tr>
<tr>
<td>Nested Object Type</td>
<td>C name of the nested object type</td>
</tr>
<tr>
<td>Nested Table</td>
<td>OCIArray *</td>
</tr>
<tr>
<td>NUMBER, NUMBER(N), NUMBER(N,N)</td>
<td>OCINumber</td>
</tr>
<tr>
<td>NUMERIC, NUMERIC(N), NUMERIC(N,N)</td>
<td>OCINumber</td>
</tr>
<tr>
<td>RAW(N)</td>
<td>OCIRaw *</td>
</tr>
<tr>
<td>REAL</td>
<td>OCINumber</td>
</tr>
<tr>
<td>REF</td>
<td>OCIRaw *</td>
</tr>
<tr>
<td>VARCHAR(N)</td>
<td>OCIString *</td>
</tr>
<tr>
<td>VARCHAR2(N), NVARCHAR2(N)</td>
<td>OCIString *</td>
</tr>
<tr>
<td>VARRAY</td>
<td>OCIArray *</td>
</tr>
</tbody>
</table>

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” on page 15-10.

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.

**OTT Type Mapping Example**

Example 15–9 demonstrates the various type mappings created by OTT when given the database types shown in Example 15–8.

**Example 15–8 Object Type Definitions for the OTT Type Mapping Example**

```sql
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));
```

The intype file includes the following:

```sql
CASE = LOWER
TYPE many_types
```

OTT generates the C structs shown in Example 15–9.

---

**Note:** Comments are provided in Example 15–9 to help explain the structs. These comments are not part of actual OTT output.

---

**Example 15–9 Various Type Mappings Created by OTT from Object Type Definitions**

```sql
#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE
#ifndef OCI_ORACLE
#include <oci.h>
#endif

typedef OCIRef many_types_ref;
typedef OCIRef object_type_ref;
```
typedef OCIArray my_varray;  /* used in many_types */
typedef OCITable my_table;   /* used in many_types*/
typedef OCIRef other_type_ref;
struct object_type          /* used in many_types */
{
    OCIString * object_name;
};
typedef struct object_type object_type;

struct object_type_ind       /* indicator struct for*/
{
    /*object_types*/
    OCIInd _atomic;
    OCIInd object_name;
};
typedef struct object_type_ind object_type_ind;

struct many_types
{
    OCIString * the_varchar;
    OCIString * the_char;
    OCIBlobLocator * the_blob;
    OCIClobLocator * the_clob;
    struct object_type the_object;
    other_type_ref * another_ref;
    many_types_ref * the_ref;
    my_varray * the_varray;
    my_table * the_table;
    OCIDate the_date;
    OCINumber the_num;
    OCIRaw * the_raw;
};
typedef struct many_types many_types;

struct many_types_ind        /* indicator struct for*/
{
    /*many_types*/
    OCIInd _atomic;
    OCIInd the_varchar;
    OCIInd the_char;
    OCIInd the_blob;
    OCIInd the_clob;
    struct object_type_ind the_object;  /* nested*/
    OCIInd another_ref;
    OCIInd the_ref;
    OCIInd the_varray;
    OCIInd the_table;
    OCIInd the_date;
    OCIInd the_num;
    OCIInd the_raw;
};
typedef struct many_types_ind many_types_ind;

#endif

Notice that 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" on page 15-24 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 \texttt{REF} in an object type attribute. For example, although the \texttt{many_types} type contains the attribute \texttt{another_ref REF other_type}, a declaration of struct \texttt{other_type} was not generated.

This example also illustrates how typedefs are used to declare \texttt{varray}, \texttt{nested table}, and \texttt{REF} types.

The typedefs occur near the beginning:

\begin{verbatim}
typedef OCIRef many_types_ref;
typedef OCIRef object_type_ref;
typedef OCIArray my_varray;
typedef OCITable my_table;
typedef OCIRef other_type_ref;
\end{verbatim}

In the struct \texttt{many_types}, the \texttt{varray}, \texttt{nested table}, and \texttt{REF} attributes are declared:

\begin{verbatim}
struct many_types
{
    ... 
    other_type_ref * another_ref;
    many_types_ref * the_ref;
    my_varray * the_varray;
    my_table * the_table;
    ... 
}
\end{verbatim}

\textbf{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 15–9.

\begin{verbatim}
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;
\end{verbatim}

The layout of the NULL struct is important. The first element in the struct (_\texttt{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 \texttt{object_type} attribute), the indicator entry for that attribute is the NULL indicator struct (\texttt{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" on page 11-21 for more information about atomic nullity

### OTT Support for Type Inheritance

To support type inheritance of objects, OTT generates a C struct to represent an object subtype by declaring the inherited attributes in an encapsulated struct with the special name "_super", before declaring the new attributes. Thus, for an object subtype that inherits from a supertype, the first element in the struct is named "_super", followed by elements corresponding to each attribute of the subtype. The type of the element named "_super" is the name of the supertype.

For example, suppose that you have a type Person_t, with subtype Student_t and subtype Employee_t, as shown in Example 15–10.

**Example 15–10  Object Type and Subtype Definitions**

```sql
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));
```

Suppose that you also have an intype file with the content shown in Example 15–11.

**Example 15–11  Contents of the Intype File**

```plaintext
CASE=SAME
TYPE EMPLOYEE_T
TYPE STUDENT_T
TYPE PERSON_T
```

Then, OTT generates the C structs for Person_t, Student_t, and Employee_t, and their NULL indicator structs, as shown in Example 15–12.

**Example 15–12  OTT Generates C Structs for the Types and Null Indicator Structs**

```c
#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
```
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:

```c
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).
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:

```sql
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:

```c
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:

```c
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.

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 15–13, which was used in Example 15–5.

```
CASE=LOWER
TYPE employee
  TRANSLATE SALARY$ AS salary
  DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE 'Person'
TYPE PURCHASE_ORDER AS p_o
```

Example 15–13 Contents of an Intype File
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 15–14 shows what the outtype file might look like after running OTT.

**Example 15–14 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
```

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.

**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 15–1 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 15–1 Using OTT with OCI**

**Accessing and Manipulating Objects with OCI**

Within the application, the OCI program can perform bind and define operations using program variables declared to be of types 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 `OCIStr`ing string.
Using OTT with OCI Applications

- 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.

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 15–15.

**Example 15–15  Content of an Intype File Named ex2c.typ**

```plaintext
TYPE BREN.PERSON
TYPE BREN.ADDRESS
```

Then you invoke OTT from the command line and specify the initialization function, as shown in Example 15–16.

**Example 15–16  Invoking OTT and Specifying the Initialization Function**

```plaintext
ott userid=bren intype=ex2c outtype=ex2co hfile=ex2ch.h initfile=ex2cv.c
```

OTT generates the `ex2cv.c` file with the contents shown in Example 15–17.
Example 15–17  Content of an OTT-Generated File Named ex2cv.c

```c
#ifndef OCI_ORACLE
#include <oci.h>
#endif

sword ex2cv(OCIEnv *env, OCIError *err)
{
    sword status = OCITypeVTInit(env, err);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
                                'BREN', 5,
                                'PERSON', 6,
                                '$8.0%', 4);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
                                'BREN', 5,
                                'ADDRESS', 7,
                                '$8.0%', 4);
    return status;
}
```

The function `ex2cv()` creates the type version table and inserts the types `BREN.PERSON` and `BREN.ADDRESS`.

If a program explicitly creates an environment handle, all the initialization functions must be generated, compiled, and linked, because they must be called for each explicitly created handle. If a program does not explicitly create any environment handles, initialization functions are not required.

A program that uses an OTT-generated header file must also use the initialization function generated at the same time. When a header file is generated by OTT and an environment handle is explicitly created in the program, then the implementation file must also be compiled and linked into the executable.

Tasks of the Initialization Function

The C initialization function supplies version information about the types processed by OTT. It adds to the type-version table the name and version identifier of every OTT-processed object 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.

---

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

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.

**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]
```
Using the Object Type Translator with OCI

Using the Object Type Translator with OCI

[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. See the parameter descriptions in "OTT Parameters" on page 15-21 for more information.

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.

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.

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.
**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.

See Also: "Structure of the Intype File" on page 15-25 for a description of the format of the intype file

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.

**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.

**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.

**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
```

### INITFUNC

The INITFUNC parameter is only used in OCI programs. It specifies the name of the initialization function generated by OTT. If this parameter is omitted, the name of the initialization function is derived from the name of the INITFILE.

```
initfunc=filename
```

### HFILE

The 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
```

### 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.

---

### 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
```
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.

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" on page 15-29

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.

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:
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.

### 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.

### 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.

**See Also:**  "Outtype File" on page 15-15 for an example of a simple user-defined intype file, and of the full outtype file that OTT generates from it
**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.

See Also: "Default Name Mapping" on page 15-30

- `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.

**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 15–18.

**Example 15–18  Object Type Definition to Demonstrate How OTT Generates Include Files**

```sql
create type px1 AS OBJECT (col1 number, col2 integer);
create type px2 AS OBJECT (col1 px1);
create type px3 AS OBJECT (col1 px1);
```

The intype file content is shown in Example 15–19.

**Example 15–19  Content of the Intype File**

```sql
CASE=lower
type px1
  hfile tott95a.h
type px3
  hfile tott95b.h
```

If you invoke OTT with the command shown in Example 15–20, then it generates the header files shown in Example 15–21 and Example 15–22.

**Example 15–20  Invoking OTT from the Command Line**

```bash
ott scott tott95i.typ outtype=tott95o.typ code=c
```

The content of the header file `tott95b.h` is shown in Example 15–21.

**Example 15–21  Content of the Header File tott95b.h**

```c
#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

The content of the header file tott95a.h is shown in Example 15–22.

Example 15–22  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

In Example 15–21, 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 15–23.

Example 15–23  Construct to Use to Conditionally Include the Header File tott95b.h

#ifndef TOTT95B_ORACLE
#include "tott95b.h"
#endif

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.
SCHEMA_NAMES Usage

This parameter affects whether the name of a type from the default schema to which OTT is connected is qualified with a schema name in the outtype file.

The name of a type from a schema other 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.

Example: Schema_Names Usage

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      VARCHAR30,
   state     CHAR(2),
   zip_code  CHAR(10) );

CREATE TYPE Person AS OBJECT
   ( name      CHAR20,
   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.

**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.

**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:

```c
#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.
This chapter begins to describe the OCI relational functions for C. It includes information about calling OCI functions in your application, along with detailed descriptions of each function call.

**See Also:** For code examples, see the demonstration programs included with your Oracle Database installation. For additional information, see Appendix B.

This chapter contains these topics:

- Introduction to the Relational Functions
- Connect, Authorize, and Initialize Functions
- Handle and Descriptor Functions
- Bind, Define, and Describe Functions

**Introduction to the Relational Functions**

This chapter and Chapter 17 describe the OCI relational function calls and cover the functions in the basic OCI.

**See Also:** "Error Handling in OCI" on page 2-20 for information about return codes and error handling

**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 Table 16–1.
### Table 16–1 Mode of a Parameter

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>A parameter that passes data to the OCI</td>
</tr>
<tr>
<td>OUT</td>
<td>A parameter that receives data from the OCI on this call</td>
</tr>
<tr>
<td>IN/OUT</td>
<td>A parameter that passes data on the call and receives data on the return from this or a subsequent call</td>
</tr>
</tbody>
</table>

**Comments**

More detailed information about the function (if available). This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

**Returns**

This optional section describes the possible values that can be returned. It can be found either before or after the Comments section.

**Example**

A complete or partial code example demonstrating the use of the function call being described. Not all function descriptions include an example.

**Related Functions**

A list of related function calls.

**Calling OCI Functions**

Unlike earlier versions of 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

**Server Round-Trips for LOB Functions**

For a table showing the number of server round-trips required for individual OCI LOB functions, see Appendix C.
## Connect, Authorize, and Initialize Functions

Table 16–1 describes the OCI connect, authorize, and initialize functions that are described in this section.

### Table 16–2  Connect, Authorize, and Initialize Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIAppCtxClearAll()&quot; on page 16-4</td>
<td>Clear all attribute-value information in a namespace of an application context</td>
</tr>
<tr>
<td>&quot;OCIAppCtxSet()&quot; on page 16-5</td>
<td>Set an attribute and its associated value in a namespace of an application context</td>
</tr>
<tr>
<td>&quot;OCIConnectionPoolCreate()&quot; on page 16-7</td>
<td>Initialize the connection pool</td>
</tr>
<tr>
<td>&quot;OCIConnectionPoolDestroy()&quot; on page 16-9</td>
<td>Destroy the connection pool</td>
</tr>
<tr>
<td>&quot;OCIDBShutdown()&quot; on page 16-10</td>
<td>Shut down Oracle Database</td>
</tr>
<tr>
<td>&quot;OCIDBStartup()&quot; on page 16-12</td>
<td>Start an Oracle Database instance</td>
</tr>
<tr>
<td>&quot;OCIEnvCreate()&quot; on page 16-13</td>
<td>Create and initialize an OCI environment handle</td>
</tr>
<tr>
<td>&quot;OCIEnvNlsCreate()&quot; on page 16-17</td>
<td>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.</td>
</tr>
<tr>
<td>&quot;OCILogoff()&quot; on page 16-21</td>
<td>Release a session that was retrieved using OCILogon2() or OCILogon()</td>
</tr>
<tr>
<td>&quot;OCILogon()&quot; on page 16-22</td>
<td>Simplify single-session logon</td>
</tr>
<tr>
<td>&quot;OCILogon2()&quot; on page 16-24</td>
<td>Create a logon session in various modes</td>
</tr>
<tr>
<td>&quot;OCIHandleAttach()&quot; on page 16-27</td>
<td>Attach to a server; initialize server context handle</td>
</tr>
<tr>
<td>&quot;OCIHandleDetach()&quot; on page 16-29</td>
<td>Detach from a server; uninitialized server context handle</td>
</tr>
<tr>
<td>&quot;OCISessionBegin()&quot; on page 16-30</td>
<td>Authenticate a user</td>
</tr>
<tr>
<td>&quot;OCISessionEnd()&quot; on page 16-33</td>
<td>Terminate a user session</td>
</tr>
<tr>
<td>&quot;OCISessionGet()&quot; on page 16-34</td>
<td>Get a session from a session pool</td>
</tr>
<tr>
<td>&quot;OCISessionPoolCreate()&quot; on page 16-40</td>
<td>Initialize a session pool</td>
</tr>
<tr>
<td>&quot;OCISessionPoolDestroy()&quot; on page 16-43</td>
<td>Destroy a session pool</td>
</tr>
<tr>
<td>&quot;OCISessionRelease()&quot; on page 16-44</td>
<td>Release a session</td>
</tr>
<tr>
<td>&quot;OCITerminate()&quot; on page 16-46</td>
<td>Detach from a shared memory subsystem</td>
</tr>
</tbody>
</table>
OCIAppCtxClearAll()

Purpose

Clears all attribute-value information in a namespace of an application context.

Syntax

```c
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 Functions

OCIAppCtxSet()
OCIAppCtxSet()

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 Functions

OCIAppCtxClearAll()
OCIConnectionPoolCreate()

Purpose

Initializes the connection pool.

Syntax

```c
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, OCICConnectionPoolCreate() 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 OCICConnectionPoolCreate() 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.

**See Also:** "Connection Pool Handle Attributes" on page A-22

**Related Functions**
OCICConnectionPoolDestroy(), OCILogon2(), OCIServerAttach()
OCIConnectionPoolDestroy()

Purpose

Destroys the connection pool.

Syntax

```c
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 Functions

OCIConnectionPoolCreate()
OCIDBShutdown()

Purpose

Shuts down an Oracle Database instance.

Syntax

```c
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.

   See Also: "Database Startup and Shutdown" on page 10-17

Related Functions

`OCIAttrSet()`, `OCIDBStartup()`
OCIDBStartup()

Purpose

Starts an Oracle Database instance.

Syntax

```c
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_DBSSTARTUPFLAG_RESTRICT` - Allows database access only to users with both the `CREATE SESSION` and `RESTRICTED SESSION` privileges (normally, the DBA).
  `OCI_DBSSTARTUPFLAG_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_DBSSTARTUPFLAG_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.

**See Also:** "Database Startup and Shutdown" on page 10-17

Related Functions

- `OCIAttrSet()`, `OCIDBShutdown()`, `OCIServerAttach()`, `OCISessionBegin()`
**OCIEnvCreate()**

**Purpose**

Creates and initializes an environment handle for OCI functions to work under.

**Syntax**

```c
sword OCIEnvCreate   ( OCIEnv        **envhpp,
                       ub4           mode,
                       const void    *ctxp,
                       const void    (*)(void  *ctxp,
                                         size_t size),
                       const void    (*)(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" on page 9-33

- **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_NEW_LENGTH_SEMANTICS** - Byte-length semantics is used consistently for all handles, regardless of character sets.
- **OCI_SUPPRESS_NLS_VALIDATION** - Suppresses NLS character validation; NLS character validation suppression is on by default beginning with Oracle Database 11g 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, 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 the user wants to initialize an environment as `OCI_THREADED`, then all libraries that are used by OCI are also initialized in the threaded mode.

If N’ substitution is turned on, the `OCIlStmtPrepare()` or `OCIlStmtPrepare2()` 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:

```bash
export ORA_NCHAR_LITERAL_REPLACE=TRUE
```

To turn on N’ substitution in `csh` shell:

```bash
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" on page 2-12 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 11g Release 1 (11.1) or higher, 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 11g Release 1 (11.1), then OCI does not allow partial characters.

The `OCI_ENABLE_NLS_VALIDATION` mode, which was the default until Oracle Database 10g 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

Example 16–1 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,
    (size_t)0, (void **)0))
{
    printf("Failed: OCIEnvCreate()\n");
    return 1;
}
...

Related Functions

OCIHandleAlloc(), OCIHandleFree(), OCIEnvNlsCreate(), OCITerminate()
OCIEnvNlsCreate()

Purpose

Creates and initializes an environment handle for OCI functions to work under. It is an enhanced version of the OCIEnvCreate() function.

Syntax

```c
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           charset );
```

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_UBC** - 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" on page 9-33

- **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
OCIEnvNlsCreate()

11g 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 specified by the user.

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()
- OCIBindByPos()
- OCIBindDynamic()
- OCIDefineByPos()
- OCIDefineDynamic()

This function enables you to set charset and ncharset IDs at environment creation time. It is an enhanced version of the OCIEnvCreate() function.

This function sets nonzero charset and ncharset as client-side database and national character sets, replacing the ones specified by NLS_LANG and NLS_NCHAR. When charset and ncharset are 0, 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 the user wants to initialize an environment as OCI_THREADED, then all libraries that are used by OCI are also initialized in the threaded mode.

If N’ substitution is turned on, the OCIStmtPrepare() or 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:

```bash
export ORA_NCHAR_LITERAL_REPLACE=TRUE
```
To turn on N’ substitution in csh shell:

```bash
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" on page 2-12 for more information about the `xtramemsz` parameter and user memory allocation
- "OCIEnvCreate()" on page 16-13 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 11g Release 1 (11.1) or higher, 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 11g Release 1 (11.1), then OCI does not allow partial characters.

The `OCI_ENABLE_NLS_VALIDATION` mode, which was the default until Oracle Database 10g 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 Functions**
- `OCIHandleAlloc()`, `OCIHandleFree()`, `OCITerminate()`, `OCINlsEnvironmentVariableGet()`
OCILogoff()

Purpose

Releases a session that was retrieved using OCILogon2() or OCILogon().

Syntax

```c
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.

See Also:  "Application Initialization, Connection, and Session Creation" on page 2-14 for more information about logging on and off in an application

Related Functions

OCILogon(), OCILogon2()
OCILogon()

**Purpose**

Creates a simple logon session.

**Syntax**

```c
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.
Connect, Authorize, and Initialize Functions

Note: Users requiring more complex sessions, such as TP monitor applications, should see "Application Initialization, Connection, and Session Creation" on page 2-14.

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 Functions

OCILogoff()
OCILogon2()

Purpose

Gets a session. This session may be a new one with a new underlying connection, or one that is started over a virtual connection from an existing connection pool, or one from an existing session pool. The mode that the function is called with determines its behavior.

Syntax

```c
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 the user 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. The user is 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. The user is given a session that is authenticated by the user name provided in the OCILogon2() call, through the proxy credentials supplied in the OCIInputStreamPoolCreate() call.

Comments

None.

Related Functions

OCILogon(), OCILogoff(), OCIInputStreamGet(), OCIInputStreamRelease()
OCI Server Attach

Purpose

Creates an access path to a data source for OCI operations.

Syntax

```c
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.

**Example**

Example 16–2 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.

**Example 16–2 Using the OCIServerAttach() Call**

```c
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 Functions**

`OCIServerDetach()`
OCI Server Detach() (OCIServerDetach)

Purpose

Deletes an access path to a data source for OCI operations.

Syntax

\[
\text{sword OCIServerDetach ( OCIServer } \* \text{srvhp,} \\
\text{OCIError } \* \text{errhp,} \\
\text{ub4 } \text{mode }) \;
\]

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 to a data source for OCI operations. The access path was established by a call to OCIServerAttach().

Related Functions

OCIServerAttach()
OCI_SessionBegin()

Purpose

Creates a user session and begins a user session for a given server.

Syntax

```c
sword OCI_SessionBegin ( 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, the user is authenticated for SYSDBA access.
- OCI_SYSOPER - In this mode, the user is authenticated for SYSOPER access.
Connect, Authorize, and Initialize Functions

- **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.

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 ushrp 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 the user gets 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

**Example 16-3** 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.

```c
/* 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 Functions

- OCISessionEnd()
OCISessionEnd()

Purpose

Terminates a user session context created by OCISessionBegin().

Syntax

sword OCISessionEnd ( OCISvcCtx       *svchp,
              OCIError        *errhp,
              OCISession      *usrhp,
              ub4             mode );

Parameters

svchp (IN/OUT)
The service context handle. There must be a valid server handle and user session handle associated with svchp.

errhp (IN/OUT)
An error handle 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().

Related Functions

OCISessionBegin()
OCISessionGet()

Purpose

Gets a session. This session may be a new one with a new underlying connection, or one that is started over a virtual connection from an existing connection pool, or one from an existing session pool. The mode that the function is called with determines its behavior.

Syntax

```
sword OCISessionGet ( OCIenv *envhp,
                     OCIError *errhp,
                     OCISvcCtx **svchp,
                     OCIAuthInfo *authInfop,
                     OraText *dbName,
                     ub4 dbName_len,
                     const OraText *tagInfo,
                     ub4 tagInfo_len,
                     OraText **retTagInfo,
                     ub4 *retTagInfo_len,
                     boolean *found,
                     ub4 mode );
```

Parameters

**envhp (IN/OUT)**
OCI environment handle. For connection pooling and session pooling, this should be the one that the respective pool was created in.

**errhp (IN/OUT)**
OCI error handle.

**svchp (OUT)**
Address of an OCI service context pointer. This 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 \texttt{OCI\_SPC\_HOMOGENEOUS}.

The attributes that can be set on the \texttt{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:

- \texttt{OCI\_ATTR\_USERNAME}
- \texttt{OCI\_ATTR\_PASSWORD}
- \texttt{OCI\_ATTR\_CONNECTION\_CLASS}
- \texttt{OCI\_ATTR\_PURITY}
- \texttt{OCI\_ATTR\_PROXY\_CREDENTIALS}
- \texttt{OCI\_ATTR\_DISTINGUISHED\_NAME}
- \texttt{OCI\_ATTR\_CERTIFICATE}
- \texttt{OCI\_ATTR\_INITIAL\_CLIENT\_ROLES}
- \texttt{OCI\_ATTR\_APPCTX\_SIZE}
- \texttt{OCI\_ATTR\_EDITION}
- \texttt{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 \texttt{OCISession} handle after the session has been created:

- \texttt{OCI\_ATTR\_CLIENT\_IDENTIFIER}
- \texttt{OCI\_ATTR\_CURRENT\_SCHEMA}
- \texttt{OCI\_ATTR\_MODULE}
- \texttt{OCI\_ATTR\_ACTION}
- \texttt{OCI\_ATTR\_CLIENT\_INFO}
- \texttt{OCI\_ATTR\_COLLECT\_CALL\_TIME}
- \texttt{OCI\_ATTR\_DEFAULT\_LOB\_PREFETCH\_SIZE}
- \texttt{OCI\_ATTR\_SESSION\_STATE}

**See Also:**

- "Session Pooling in OCI" on page 9-7
- \texttt{cdemosp.c} in the demo directory
- "User Session Handle Attributes" on page A-14 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 the user wants a default session, the user must set this to `NULL`. See the Comments for a detailed explanation of this parameter.

**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_SESGET_CPOOL`
- `OCI_SESGET_SPOOL`
- `OCI_SESGET_CREDDPROXY`
- `OCI_SESGET_CREDEXT` - Supported only for heterogeneous pools.
- `OCI_SESGET_PURITY_NEW`
- `OCI_SESGET_PURITY_SELF`
- `OCI_SESGET_SPOOL_MATCHANY`
- `OCI_SESGET_STMTCACHE`
- `OCI_SESGET_SYSDBA`

In the default (nonpooling) case, the following modes are valid:

- `OCI_SESGET_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 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 OCIConnectionPoolCreate() 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.

For database resident connection pooling (DRCP), the following modes are valid:

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.

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 OCIConnectionPoolCreate() 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 OCIConnectionPoolCreate() 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.

Example 16-4 demonstrates the use of OCI_ATTR_MODULE with session pooling.

Example 16-4  Using the OCI_ATTR_MODULE Attribute with OCI Session Pooling

Oratext *module = (Oratext*) "mymodule";
/* Allocate the pool handle */
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.

Example 16–5 shows how to use the OCI_ATTR_EDITION attribute with an OCI session pool.

Example 16–5 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,
    OCITSTR("EDITION"), strlen(OCITSTR("EDITION")), 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, 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, 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.

Related Functions

    OCISessionRelease(), OCISessionPoolCreate(), OCISessionPoolDestroy()
OCISessionPoolCreate()

Purpose

Initializes a session pool for use with OCI session pooling and database resident connection pooling (DRCP). 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

```c
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.

\[\text{sessMin} + \text{sessIncr} \text{ cannot be more than sessMax.}\]

userid (IN)
Specifies the userid with which to start the sessions.

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.

\[\text{See Also: "Statement Caching in OCI" on page 9-26}\]

- **OCI_SPC_HOMOGENEOUS** - All sessions in the pool are authenticated with the user name and password passed to OCISessionPoolCreate(). The authentication handle (parameter authInfo) passed into OCISessionGet() is ignored in this case. Moreover, the sessMin and the sessIncr values are considered only in this case. No proxy session can be created in this mode. 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" on page 2-15

Oracle Database SQL Language Reference
Oracle Database Concepts

When the session pool is created, the userid and password may or may not be specified. If these values are NULL, no proxy connections can exist in this pool. If mode is set to OCI_SPC_HOMOGENEOUS, no proxy connection can exist.

A userid and password pair may also be specified through the authentication handle in the OCISessionGet() call. If this call is made with mode set to OCI_SESSGET_CREDPROXY, then the user is given a session that is authenticated by the userid provided in the OCISessionGet() call, through the proxy credentials supplied in the OCISessionPoolCreate() call. In this case, the password in the OCISessionGet() call is ignored.

If OCISessionGet() is called with mode not set to OCI_SESSGET_CREDPROXY, then the user gets a direct session 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

Example 16–6 shows how to disable runtime load balancing.

Example 16–6 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 Functions
OCISessionRelease(), OCISessionGet(), OCISessionPoolDestroy()
OCISessionPoolDestroy()

Purpose

Destroys a session pool.

Syntax

```c
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 Functions

OCISessionPoolCreate(), OCISessionRelease(), OCISessionGet()
OCISessionRelease()

Purpose

Releases a session that was retrieved using OCIConnectionGet(). The exact behavior of this call is determined by the mode in which the corresponding OCIConnectionGet() 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

```c
sword OCIConnectionRelease ( OCISvcCtx       *svchp,
                              OCIError        *errhp,
                              OraText         *tag,
                              ub4             tag_len,
                              ub4             mode );
```

Parameters

**svchp (IN)**
The service context that was populated during the corresponding OCIConnectionGet() 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_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.

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 Functions

- `OCISessionGet()`, `OCISessionPoolCreate()`, `OCISessionPoolDestroy()`, `OCILogon2()`
OCI Terminate()

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 Functions

OCIInitialize(), OCIEnvCreate(), OCIEnvNlsCreate()
Handle and Descriptor Functions

Table 16–3 lists the OCI handle and descriptor functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIArrayDescriptorAlloc()&quot; on page 16-48</td>
<td>Allocate an array of descriptors</td>
</tr>
<tr>
<td>&quot;OCIArrayDescriptorFree()&quot; on page 16-50</td>
<td>Free an array of descriptors</td>
</tr>
<tr>
<td>&quot;OCIAttrGet()&quot; on page 16-51</td>
<td>Get the value of an attribute of a handle</td>
</tr>
<tr>
<td>&quot;OCIAttrSet()&quot; on page 16-53</td>
<td>Set the value of an attribute of a handle or descriptor</td>
</tr>
<tr>
<td>&quot;OCIDescriptorAlloc()&quot; on page 16-54</td>
<td>Allocate and initialize a descriptor or LOB locator</td>
</tr>
<tr>
<td>&quot;OCIDescriptorFree()&quot; on page 16-56</td>
<td>Free a previously allocated descriptor</td>
</tr>
<tr>
<td>&quot;OCIHandleAlloc()&quot; on page 16-57</td>
<td>Allocate and initialize a handle</td>
</tr>
<tr>
<td>&quot;OCIHandleFree()&quot; on page 16-58</td>
<td>Free a previously allocated handle</td>
</tr>
<tr>
<td>&quot;OCIParamGet()&quot; on page 16-59</td>
<td>Get a parameter descriptor</td>
</tr>
<tr>
<td>&quot;OCIParamSet()&quot; on page 16-61</td>
<td>Set parameter descriptor in COR handle</td>
</tr>
</tbody>
</table>
OCIArrayDescriptorAlloc()

Purpose

Allocates an array of descriptors.

Syntax

```c
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()" on page 16-54.

**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 the user memory of size xtramem_sz allocated by the call for the user for the lifetime of the descriptors.

Comments

This call returns OCI_SUCCESS if successful, or a suitable error if an out-of-memory condition occurs.

**See Also:** "User Memory Allocation" on page 2-12 for more information about the xtramem_sz parameter and user memory allocation

Example

Example 16–7 can be modified to allocate a large number of descriptors.

**Example 16–7  Allocating a Large Number of Descriptors**

```c
OCIDateTime *descpl[500];
...
for (i = 0; i!=500; i++)
{
```
/* Allocate descriptors */
OCIDescriptorAlloc((void *)envhp,(void **)&descppl[i], OCI_DTYPE_TIMESTAMP_TZ,
                  0,(void **)0);
}
...

The for loop in Example 16–7 can now be replaced with a single call, as shown in Example 16–8.

Example 16–8  Allocating an Array of Descriptors

OCIArrayDescriptorAlloc((void *)envhp,(void **)&descpl,
                         OCI_DTYPE_TIMESTAMP_TZ, 500, 0, (void **)0);

Related Functions

OCIDescriptorAlloc(), OCIArrayDescriptorFree()
OCIArrayDescriptorFree()

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()"
on page 16-54.

Comments
An error is returned when a descriptor is allocated using OCIDescriptorAlloc(), but
freed using OCIArrayDescriptorFree().

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 Functions
OCIArrayDescriptorAlloc()
OCIAttrGet()

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 2–1 or any descriptor in Table 2–2.

attributep (OUT)
Pointer to the storage for an attribute value. 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 Appendix A.

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 \texttt{OCIAttrGet()} to get the Unicode information in an environment or statement handle.

\textbf{See Also:} \textit{"Examples Using OCIDescribeAny()"} on page 6-18 and \textit{"Describing Select-List Items"} on page 4-9

A function closely related to \texttt{OCIAttrGet()} is \texttt{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 \texttt{OCIAttrGet()} call.

Then an \texttt{OCIParamGet()} call on the describe handle returns a parameter descriptor for a specified position. Parameter positions begin with 1. Calling \texttt{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 \texttt{OCIDescribeAny()}. Calling \texttt{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 \texttt{OCIAttrGet()} is called. If multiple pointers are used for the same attribute, a memory leak can occur.

\textbf{Related Functions}

\texttt{OCIAttrSet()}

\texttt{OCIAttrGet()}

\texttt{OCIDescribeAny()}

\texttt{OCIParamGet()}
**OCIAttrSet()**

**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 Appendix A for a list of handle types and their writable attributes.

**Related Functions**

- `OCIArrayDescriptorAlloc()`
OCIDescriptorAlloc()

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_AQNOTIFY - 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 OCIAQLisOpts.

OCI_DTYPE_AQLIS_MSG_PROPERTIES - Specifies generation of an Advanced Queuing message properties descriptor of C type OCIAQLisMsgProps.

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.

**See Also:** "User Memory Allocation" on page 2-12 for more information about the xtramem_sz parameter and user memory allocation.

**Related Functions**

OCIDescriptorFree(), OCIArrayDescriptorAlloc(), OCIArrayDescriptorFree()
OCIDescriptorFree()  

Purpose  
Deallocates a previously allocated descriptor.

Syntax  
```c
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()” on page 16-54.

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.

Related Functions  
OCIDescriptorAlloc()
OCIHandleAlloc()

Purpose

Returns a pointer to an allocated and initialized handle.

Syntax

```c
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 2–1.

- `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.

See Also: "User Memory Allocation" on page 2-12 for more information about using the `xtramem_sz` parameter for user memory allocation

Related Functions

- `OCIHandleFree()`
OCIHandleFree()

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 2-1.

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 Functions
OCIHandleAlloc()
OCIParamGet()

Purpose
Returns a descriptor of a parameter specified by position in the describe handle or statement handle.

Syntax

```c
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().

See Also: Appendix A for more detailed information about parameter descriptor attributes
Related Functions

OCIArrayDescriptorAlloc(), OCIAttrSet(), OCIParamSet(), OCI DescriptorFree()
**OCIParamSet()**

**Purpose**
Sets a complex object retrieval (COR) descriptor into a COR handle.

**Syntax**
```c
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().

**See Also:** "Complex Object Retrieval" on page 11-15 for more information about complex object retrieval

**Related Functions**
OCIParamGet()
Bind, Define, and Describe Functions

Table 16–4 lists the bind, define, and describe functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIBindArrayOfStruct()&quot; on page 16-63</td>
<td>Set skip parameters for static array bind</td>
</tr>
<tr>
<td>&quot;OCIBindByName()&quot; on page 16-64</td>
<td>Bind by name</td>
</tr>
<tr>
<td>&quot;OCIBindByPos()&quot; on page 16-68</td>
<td>Bind by position</td>
</tr>
<tr>
<td>&quot;OCIBindDynamic()&quot; on page 16-72</td>
<td>Set additional attributes after bind with OCI_DATA_AT_EXEC mode</td>
</tr>
<tr>
<td>&quot;OCIBindObject()&quot; on page 16-75</td>
<td>Set additional attributes for bind of named data type</td>
</tr>
<tr>
<td>&quot;OCIDefineArrayOfStruct()&quot; on page 16-77</td>
<td>Set additional attributes for static array define</td>
</tr>
<tr>
<td>&quot;OCIDefineByPos()&quot; on page 16-78</td>
<td>Define an output variable association</td>
</tr>
<tr>
<td>&quot;OCIDefineDynamic()&quot; on page 16-82</td>
<td>Set additional attributes for define in OCI_DYNAMIC_FETCH mode</td>
</tr>
<tr>
<td>&quot;OCIDefineObject()&quot; on page 16-84</td>
<td>Set additional attributes for define of named data type</td>
</tr>
<tr>
<td>&quot;OCIDescribeAny()&quot; on page 16-86</td>
<td>Describe existing schema objects</td>
</tr>
<tr>
<td>&quot;OCIStmtGetBindInfo()&quot; on page 16-89</td>
<td>Get bind and indicator variable names and handle</td>
</tr>
</tbody>
</table>
OCIBindArrayOfStruct()

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.

See Also: "Binding and Defining Arrays of Structures in OCI" on page 5-18 for information about skip parameters

Related Functions
OCIBindByName(), OCIBindByPos()
**OCIBindByName()**

**Purpose**

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.

**Syntax**

```c
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`. If not, it could be a binary string, double, integer, character, or any other data type.
Furthermore, as pointed out for \texttt{OCIStmtPrepare()}, the default encoding for the string type \texttt{valuep} is in the encoding specified by the \texttt{charset} parameter of a previous call to \texttt{OCIEnvNlsCreate()}, unless users call \texttt{OCIAttrSet()} to manually reset the character set for the bind handle.

**See Also:** "Bind Handle Attributes" on page A-34

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 \texttt{SQLT_NTY} or \texttt{SQLT_REF} binds, the \texttt{valuep} parameter is ignored. The pointers to OUT buffers are set in the \texttt{pgvpp} parameter initialized by \texttt{OCIBindObject()}. When \texttt{mode} is set to \texttt{OCI_IOV}, pass the base address of the \texttt{OCIIOV} struct.

\textbf{value\_sz (IN)}

The maximum size possible in bytes of any data value (passed using \texttt{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 \texttt{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, \texttt{sizeof(OCILobLocator *)}.

The same applies even when \texttt{mode} is \texttt{OCI_IOV}.

\textbf{dty (IN)}

The data type of the values being bound. Named data types (\texttt{SQLT_NTY}) and REFs (\texttt{SQLT_REF}) are valid only if the application has been initialized in object mode. For named data types or REFs, additional calls must be made with the bind handle to set up the data type-specific attributes.

\textbf{indp (IN/OUT)}

Pointer to an indicator variable or array. For all data types except \texttt{SQLT_NTY}, this is a pointer to \texttt{sb2} or an array of \texttt{sb2}.

For \texttt{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 \texttt{OCIBindObject()}. This parameter is ignored for dynamic binds.

**See Also:** "Indicator Variables" on page 2-24

\textbf{alenp (IN/OUT)}

Pointer to the array of actual lengths of array elements.

When \texttt{OCIEnvNlsCreate()} (which is the recommended OCI environment handle creation interface) is used, then \texttt{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 \texttt{SQLT_VCS} (2-byte length prefix) and \texttt{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 \texttt{OCIEnvCreate()}) or deprecated \texttt{OCIEnvInit()}), \texttt{alenp} lengths are in bytes in general. However, \texttt{alenp} lengths are expected in characters for IN binds and also reported in characters for OUT binds only when either the character set is \texttt{OCI_UC2ID} (= \texttt{OCL_UTF16ID}) or when \texttt{OCI_ATTR_CHAR_COUNT} attribute is set on the corresponding \texttt{OCIBind} handle. The same treatment holds for the length prefix in \texttt{SQLT_VCS} (2-byte length prefix) and \texttt{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 OCIStmtPrepare(). Because the encoding of bind variables should always be same as that of the statement containing this variable, an error is raised if the user specifies 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 stmp 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 sd 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" on page 5-33 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 value parameter must be of the type **OCIIOV**.

**See Also:** "Binding and Defining Multiple Buffers" on page 5-20

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() also implicitly allocates the bind handle indicated by the bindpp parameter. If a non-NULL pointer is passed in *bindpp, the OCI assumes that this points to a valid handle that has been previously allocated with a call to OCIHandleAlloc() or OCIBindByName().

Data in an OCI application can be bound to placeholders statically or dynamically. Binding is static when all the IN bind data and the OUT bind buffers are well defined just before the execute. Binding is dynamic when the IN bind data and the OUT bind buffers are provided by the application on demand at execute time to the client library. Dynamic binding is indicated by setting the mode parameter of this call to OCI_DATA_AT_EXEC.

See Also: "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 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 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.

Related Functions

OCIBindDynamic(), OCIBindObject(), OCIBindArrayOfStruct()
OCIBindByPos()

Purpose

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.

Syntax

```
sword OCIBindByPos ( OCIStmt      *stmtp,
                   OCIBind      **bindpp,
                   OCIError     *errhp,
                   ub4          position,
                   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.

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.
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 REFS (SQLT_REF) are valid only if the application has been initialized in object mode. For named data types or REFS, additional calls must be made with the bind handle to set up the attributes specific to the data type.

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" on page 2-24

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" on page 5-33 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 *`.

  **See Also:** "Binding and Defining Multiple Buffers" on page 5-20

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 execute time to the
client library. Dynamic binding is indicated by setting the `mode` parameter of this call to `OCI_DATA_AT_EXEC`.

**See Also:** "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 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.

**Related Functions**

`OCIBindDynamic()`, `OCIBindObject()`, `OCIBindArrayOfStruct()`
OCIBindDynamic()

Purpose

Registers user callbacks for dynamic data allocation.

Syntax

```c
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 REFs, pass the address of the ref; that is, pass &my_ref for *bufpp.

If the OCI_ATTR_CHARSET_ID attribute is set to OCI_UTF16ID (replaces the deprecated OCI_UCS2ID, which is retained for backward compatibility), all data passed to and received with the corresponding bind call is assumed to be in UTF-16 encoding.

See Also:  "Bind Handle Attributes" on page A-34

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" on page A-34.

**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" on page 5-33 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 Functions**

`OCIBindByName()`, `OCIBindByPos()`
OCIBindObject()

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 OCIBindByPos().

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 REFS in SQL, but required for REFS in PL/SQL.

pgvpp (IN/OUT)
Address of the program variable buffer. For an array, pgvpp points to an array of addresses. When the bind variable is also an OUT variable, the OUT named data type value or REF is allocated in the Object Cache, and a REF is returned.

pgvpp is ignored if the OCI_DATA_AT_EXEC mode is set. Then the named data type buffers are requested at 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 ub4s. On return, for OUT bind variables, this points to sizes of the named data types and REFS received. pvszsp is ignored if the OCI_DATA_AT_EXEC mode is set. Then the size of the buffer is taken at 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 sb2s. 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 **OCIBindByPos()**, the pointers to the IN buffers are obtained either using the callback **icbfp** registered in the **OCIBindDynamic()** call or by the **OCIStmtSetPieceInfo()** call.

The buffers are dynamically allocated for the OUT data. The pointers to these buffers are returned either by:

- Calling **ocbfp()** registered by the **OCIBindDynamic()**
- Setting the pointer to the buffer in the buffer passed in by **OCIStmtSetPieceInfo()** called when **OCIStmtExecute()** returned **OCI_NEED_DATA**

The memory of these client library-allocated buffers must be freed when not in use anymore by using the **OCIObjectFree()** call.

**Related Functions**
**OCIBindByName()**, **OCIBindByPos()**
**OCIDefineArrayOfStruct()**

**Purpose**
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()`.
- **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()`. If the application is binding an array of structures involving objects, it must call `OCIDefineObject()` first, and then call `OCIDefineArrayOfStruct()`.

**See Also:** "Skip Parameters" on page 5-18

**Related Functions**
- `OCIDefineByPos()`, `OCIDefineObject()`
OCIDefineByPos()

Purpose

Associates an item in a select list with the type and output data buffer.

Syntax

```c
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:** The user must keep track of this pointer. If a second call to OCIDefineByPos() is made for the same column position, there is no guarantee that the same pointer 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, may
be additionally 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" on page A-34

**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:** Chapter 3 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" on page 2-24

**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 OCIHandle. 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_DEFINE_FETCH - For applications requiring dynamically allocated data at the
time of fetch, this mode must be used. You can define a callback using the
OCI_DefineDynamic() 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" on page 10-20

OCI_IOV - Define noncontiguous addresses of data. The valuep parameter must be
of the type OCIIOV *.

See Also:  "Binding and Defining Multiple Buffers" on page 5-20

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 OCI_DefineByPos(). 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 OCI_DefineByPos(), which defines the minimal attributes required to specify the
fetch.

Following the call to OCI_DefineByPos() additional define calls may be necessary for
certain data types or fetch modes:

A call to OCI_DefineArrayOfStruct() is necessary to set up skip parameters for an
array fetch of multiple columns.

A call to OCI_DefineObject() 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 OCI_DefineByPos() is ignored.

Both OCI_DefineArrayOfStruct() and OCI_DefineObject() must be called after
OCI_DefineByPos() 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
OCI_LobLocator, allocated by the OCI_DescriptorAlloc() 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 \texttt{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 \texttt{OCI\_NEED\_DATA} error code, and a piecewise polling method is used to provide the data.

\textbf{See Also:}
- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for more information about using the \texttt{OCI\_DYNAMIC\_FETCH} mode
- "Overview of Defining in OCI" on page 5-13 for more information about defines
- "Implicit Fetching of ROWIDs" on page 10-20

\textbf{Related Functions}

\texttt{OCI\_DefineArrayOfStruct()}, \texttt{OCI\_DefineDynamic()}, \texttt{OCI\_DefineObject()}
OCIDefineDynamic()

Purpose

Sets the additional attributes required if the OCI_DYNAMIC_FETCH mode was selected in OCIDefineByPos().

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.

Caution: 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_DYNAMICFETCH mode has been selected in a call to OCIDefineByPos(). If OCI_DYNAMIC_FETCH mode was selected, and the call to OCIDefineDynamic() is skipped, then the application can fetch data piecewise using OCI calls (OCIStmtGetPieceInfo() and OCIStmtSetPieceInfo()).

**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.

**See Also:** "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for more information about OCI_DYNAMIC_FETCH mode

**Related Functions**

OCIDefineObject(), OCIBindDynamic()
**OCIDefineObject()**

**Purpose**
Sets up additional attributes necessary for a named data type or `REF` define.

**Syntax**
```c
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()`.

- **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()` 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()`.

See Also:

- "OCIEnvCreate()" on page 16-13, and "OCIEnvNlsCreate()" on page 16-17 for more information about initializing the OCI process environment
- "Binding and Defining Multiple Buffers" on page 5-20 for an example of using multiple buffers

Related Functions

`OCIDefineByPos()`
OCIDescribeAny()

**Purpose**

Describes existing schema and subschema objects.

**Syntax**

```c
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 Oracle8i 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. 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.

**See Also:** Chapter 6 for more information about describe operations

**Related Functions**

`OCIArrayDescriptorAlloc()`, `OCIParamGet()`
Bind, Define, and Describe Functions

OCI Relational Functions

OCIStmtGetBindInfo()

Purpose

Gets the bind and indicator variable names.

Syntax

```c
sword OCIStmtGetBindInfo ( OCIStmt      *stmtp,
                          OCIError     *errhp,
                          ub4          size,
                          ub4          startloc,
                          sb4          *found,
                          OraText      *bvnp[
                          ub1          bvnl[
                          OraText      *invp[
                          ub1          inpl[
                          ub1          dupl[
```

Parameters

stmtp (IN)
The statement handle prepared by OCIStmtPrepare().

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 *OCIStmtPrepare()* prior to this call. The encoding setting in the statement handle determines whether Unicode strings are retrieved.

This call is processed locally.

**Related Functions**

*OCIStmtPrepare()*
This chapter 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 Appendix B.

This chapter contains these topics:

- Introduction to the Relational Functions
- Statement Functions
- LOB Functions
- Streams Advanced Queuing and Publish-Subscribe Functions
- Direct Path Loading Functions
- Thread Management Functions
- Transaction Functions
- Miscellaneous Functions

Introduction to the Relational Functions

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" on page 2-20 for information about return codes and error handling

Conventions for OCI Functions

See the "Conventions for OCI Functions" on page 16-1 for the conventions used in describing each function.
Table 17–1 lists the statement functions that are described in this section. Use functions that end in "2" for all new applications.

Table 17–1  Statement Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIStmtExecute()&quot; on page 17-3</td>
<td>Send statements to server for execution</td>
</tr>
<tr>
<td>&quot;OCIStmtFetch2()&quot; on page 17-6</td>
<td>Fetch rows from a query and fetches a row from the (scrollable) result set.</td>
</tr>
<tr>
<td>&quot;OCIStmtGetPieceInfo()&quot; on page 17-8</td>
<td>Get piece information for piecewise operations</td>
</tr>
<tr>
<td>&quot;OCIStmtPrepare()&quot; on page 17-10</td>
<td>Prepare a SQL or PL/SQL statement for execution</td>
</tr>
<tr>
<td>&quot;OCIStmtPrepare2()&quot; on page 17-12</td>
<td>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.</td>
</tr>
<tr>
<td>&quot;OCIStmtRelease()&quot; on page 17-14</td>
<td>Release the statement handle</td>
</tr>
<tr>
<td>&quot;OCIStmtSetPieceInfo()&quot; on page 17-15</td>
<td>Set piece information for piecewise operations</td>
</tr>
</tbody>
</table>
OCIStmtExecute()

Purpose

Associates an application request with a server.

Syntax

```c
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, set `iters` to zero.

This function returns an error if `iters=0` for non-SELECT statements.

**Note:** For array DML operations, set `iters <= 32767` to get better performance.

**rowoff (IN)**
The starting index from which the data in an array bind is relevant for this multiple row execution.

**snap_in (IN)**
This parameter is optional. If it is supplied, 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.
**OCIStmtExecute()**

**snap_out (OUT)**
This parameter is optional. If it is supplied, 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 "Batch Error Mode" on page 4-7 for information about this mode.
- **OCI_COMMIT_ON_SUCCESS** - When a statement is executed in this mode, the current transaction is committed after execution, if execution completes successfully.
- **OCI_DEFAULT** - Calling `OCIStmtExecute()` in this mode executes the statement. It also implicitly returns describe information about the select list.
- **OCI_DESCRIBE_ONLY** - This mode is for users who 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 "Fetching Results" on page 4-13. This mode cannot be used with any other mode.

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, the description of the select list is available implicitly as a response. This description is buffered on the client side for describes, fetches, and define type conversions. Hence it is optimal to describe a select list only after an execute.

**See Also:**  "Describing Select-List Items" on page 4-9

Also for `SELECT` statements, some results are available implicitly. Rows 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 user’s responsibility to maintain the integrity of the service context. The implicit reference is maintained until the statement handle is freed or the fetch is canceled or an end of fetch condition is reached.

To reexecute a DDL statement, you must prepare the statement again using OCIStmtPrepare() or 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().

---

**See Also:** "Polling Mode Operations in OCI" on page 2-27

**Related Functions**

OCIStmtPrepare()
OCIStmtFetch2()

Purpose

Fetches a row from the (scrollable) result set. You are encouraged to use this fetch call instead of the deprecated call OCIStmtFetch().

Syntax

```c
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_FETCHPRIOR with a value of fetchOffset equal to -1

OCI_ATTR_ROW_COUNT contains the highest absolute row value that was fetched.

All other orientation modes besides OCI_FETCH_ABSOLUTE and OCI_FETCH_RELATIVE 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.

See Also:

- "Using Scrollable Cursors in OCI" on page 4-14
- "Polling Mode Operations in OCI" on page 2-27

Related Functions

OCIStmtExecute(), OCIBindByPos()
OCIStmtGetPieceInfo()

Purpose

Returns piece information for a piecewise operation.

Syntax

```c
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.
See Also:

- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for more information about using `OCIStmtGetPieceInfo()`
- "Polling Mode Operations in OCI" on page 2-27

Related Functions

OCIArrayDescriptorAlloc(), OCIAttrSet(), OCIStmtExecute(), OCIStmtFetch() (deprecated), OCIStmtFetch2(), OCIStmtSetPieceInfo()
OCIStmtPrepare()

Purpose

Prepares a SQL or PL/SQL statement for execution.

Syntax

```c
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.

mode (IN)
Similar to the mode in the OCIEnvCreate() call, but this one has higher priority because it can override the "naturally" inherited mode setting.

The only possible value is OCI_DEFAULT (default mode). The statement handle stmp 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.

**See Also:** "Preparing Statements" on page 4-3 for more information about using this call

**Related Functions**

OCIArrayDescriptorAlloc(), OCIStmtExecute(), OCIStmtPrepare2()
OCIStmtPrepare2()

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.

Syntax

```c
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 based solely 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.

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. This is the only valid setting. 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.

**Related Functions**

OCIStmtRelease()
OCIStmtRelease()

Purpose

Releases the statement handle obtained by a call to OCIStmtPrepare2().

Syntax

```c
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_STMTS_CACHE_DELETE - Only valid for statement caching. The statement is not kept in the cache anymore.

Related Functions

OCIStmtPrepare2()
OCISstmtSetPieceInfo()

Purpose

Sets piece information for a piecewise operation.

Syntax

```c
sword OCISstmtSetPieceInfo ( 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 REFS, 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 REFS (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.

See Also:
- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for more information about using `OCIStmtSetPieceInfo()`
- "Polling Mode Operations in OCI" on page 2-27

Related Functions

`OCIArrayDescriptorAlloc()`, `OCIAttrSet()`, `OCIStmtExecute()`, `OCIStmtFetch()` (deprecated), `OCIStmtFetch2()`, `OCIStmtGetPieceInfo()`
### LOB Functions

Table 17–2 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.

**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:
- "Binding LOB Data" on page 5-9 for usage and examples for both INSERT and UPDATE statements
- "Defining LOB Data" on page 5-16 for usage and examples of SELECT statements
- Chapter 7, "LOB and BFILE Operations"

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIDurationBegin()&quot; on page 17-20</td>
<td>Start user duration for temporary LOB</td>
</tr>
<tr>
<td>&quot;OCIDurationEnd()&quot; on page 17-21</td>
<td>End user duration for temporary LOB</td>
</tr>
<tr>
<td>&quot;OCILobAppend()&quot; on page 17-22</td>
<td>Append one LOB to another</td>
</tr>
<tr>
<td>&quot;OCILobArrayRead()&quot; on page 17-24</td>
<td>Read LOB data for multiple locators</td>
</tr>
<tr>
<td>&quot;OCILobArrayWrite()&quot; on page 17-28</td>
<td>Write LOB data for multiple locators</td>
</tr>
<tr>
<td>&quot;OCILobAssign()&quot; on page 17-32</td>
<td>Assign one LOB locator to another</td>
</tr>
<tr>
<td>&quot;OCILobCharSetForm()&quot; on page 17-34</td>
<td>Get character set form from LOB locator</td>
</tr>
<tr>
<td>&quot;OCILobCharSetId()&quot; on page 17-35</td>
<td>Get character set ID from LOB locator</td>
</tr>
<tr>
<td>&quot;OCILobClose()&quot; on page 17-36</td>
<td>Close a previously opened LOB</td>
</tr>
<tr>
<td>&quot;OCILobCopy2()&quot; on page 17-37</td>
<td>Copy all or part of one LOB to another. This function must be used for LOBs of size greater than 4 GB.</td>
</tr>
<tr>
<td>&quot;OCILobCreateTemporary()&quot; on page 17-39</td>
<td>Create a temporary LOB</td>
</tr>
<tr>
<td>&quot;OCILobDisableBuffering()&quot; on page 17-41</td>
<td>Turn LOB buffering off</td>
</tr>
<tr>
<td>&quot;OCILobEnableBuffering()&quot; on page 17-42</td>
<td>Turn LOB buffering on</td>
</tr>
<tr>
<td>&quot;OCILobErase2()&quot; on page 17-43</td>
<td>Erase a portion of a LOB. This function must be used for LOBs of size greater than 4 GB.</td>
</tr>
<tr>
<td>&quot;OCILobFileClose()&quot; on page 17-45</td>
<td>Close a previously opened BFILE</td>
</tr>
<tr>
<td>&quot;OCILobFileCloseAll()&quot; on page 17-46</td>
<td>Close all previously opened files</td>
</tr>
<tr>
<td>&quot;OCILobFileExists()&quot; on page 17-47</td>
<td>Check if a file exists on the server</td>
</tr>
<tr>
<td>&quot;OCILobFileGetName()&quot; on page 17-48</td>
<td>Get directory object and file name from the LOB locator</td>
</tr>
<tr>
<td>&quot;OCILobFileIsOpen()&quot; on page 17-50</td>
<td>Check if file on server is open using this locator</td>
</tr>
<tr>
<td>&quot;OCILobFileOpen()&quot; on page 17-51</td>
<td>Open a BFILE</td>
</tr>
<tr>
<td>&quot;OCILobFileSetName()&quot; on page 17-52</td>
<td>Set directory object and file name in the LOB locator</td>
</tr>
<tr>
<td>&quot;OCILobFlushBuffer()&quot; on page 17-53</td>
<td>Flush the LOB buffer</td>
</tr>
</tbody>
</table>
Note the following for parameters in the OCI LOB calls:

- For fixed-width client-side character sets, the offset and amount parameters are always in characters for CLOBs and NCLOBs, and in bytes for BLOBs and BFILES.
- For varying-width client-side character sets, these rules generally apply:
  - Amount \( \text{amtp} \) 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 \texttt{OCILOBGetLength()} (deprecated), \texttt{OCILOBGetLength2()}, \texttt{OCILOBRead()} (deprecated), \texttt{OCILOBRead2()}, \texttt{OCILOBWrite()} (deprecated), and \texttt{OCILOBWrite2()}.

- Offset \( \text{offset} \) parameter - Regardless of whether the client-side character set is varying-width, the offset parameter is always in characters for CLOBs and NCLOBs and in bytes for BLOBs and BFILES.
- For many of the LOB operations, regardless of the client-side character set, the amount parameter is in characters for CLOBs and NCLOBs. These LOB operations

---

### Table 17–2 (Cont.) LOB Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCILOBFreeTemporary()&quot; on page 17-54</td>
<td>Free a temporary LOB</td>
</tr>
<tr>
<td>&quot;OCILOBGetChunkSize()&quot; on page 17-55</td>
<td>Get the chunk size of a LOB</td>
</tr>
<tr>
<td>&quot;OCILOBGetContentType()&quot; on page 17-56</td>
<td>Retrieve the user-specified content type string (a file format identifier) for a SecureFile</td>
</tr>
<tr>
<td>&quot;OCILOBGetLength2()&quot; on page 17-58</td>
<td>Get length of a LOB. This function must be used for LOBs of size greater than 4 GB.</td>
</tr>
<tr>
<td>&quot;OCILOBGetOptions()&quot; on page 17-59</td>
<td>Get option settings of a SecureFile</td>
</tr>
<tr>
<td>&quot;OCILOBGetStorageLimit()&quot; on page 17-61</td>
<td>Get the maximum length of an internal LOB (BLOB, CLOB, or NCLOB) in bytes</td>
</tr>
<tr>
<td>&quot;OCILOBIsEqual()&quot; on page 17-62</td>
<td>Compare two LOB locators for Equality</td>
</tr>
<tr>
<td>&quot;OCILOBIsOpen()&quot; on page 17-63</td>
<td>Check to see if a LOB is open</td>
</tr>
<tr>
<td>&quot;OCILOBIsTemporary()&quot; on page 17-65</td>
<td>Determine if a given LOB is temporary</td>
</tr>
<tr>
<td>&quot;OCILOBLoadFromFile2()&quot; on page 17-66</td>
<td>Load a LOB from a BFILE. This function must be used for LOBs of size greater than 4 GB.</td>
</tr>
<tr>
<td>&quot;OCILOBLocatorAssign()&quot; on page 17-68</td>
<td>Assign one LOB locator to another</td>
</tr>
<tr>
<td>&quot;OCILOBLocatorIsInit()&quot; on page 17-70</td>
<td>Check to see if a LOB locator is initialized</td>
</tr>
<tr>
<td>&quot;OCILOBOpen()&quot; on page 17-71</td>
<td>Open a LOB</td>
</tr>
<tr>
<td>&quot;OCILOBRead2()&quot; on page 17-73</td>
<td>Read a portion of a LOB. This function must be used for LOBs of size greater than 4 GB.</td>
</tr>
<tr>
<td>&quot;OCILOBSetContentType()&quot; on page 17-77</td>
<td>Store the user-specified content type string of the SecureFile</td>
</tr>
<tr>
<td>&quot;OCILOBSetOptions()&quot; on page 17-79</td>
<td>Enable option settings for existing and newly created SecureFiles</td>
</tr>
<tr>
<td>&quot;OCILOBTrim2()&quot; on page 17-80</td>
<td>Truncate a LOB. This function must be used for LOBs of size greater than 4 GB.</td>
</tr>
<tr>
<td>&quot;OCILOBWrite2()&quot; on page 17-81</td>
<td>Write into a LOB. This function must be used for LOBs of size greater than 4 GB.</td>
</tr>
<tr>
<td>&quot;OCILOBWriteAppend2()&quot; on page 17-85</td>
<td>Write data beginning at the end of a LOB. This function must be used for LOBs of size greater than 4 GB.</td>
</tr>
</tbody>
</table>
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.
OCI Duration Begin

Purpose

Starts a user duration for a temporary LOB.

Syntax

```c
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.

See Also:  "Temporary LOB Durations" on page 7-15

Related Functions

- `OCIDurationEnd()`
LOB Functions

More OCI Relational Functions

OCIDurationEnd()

Purpose
Terminates a user duration for a temporary LOB.

Syntax
sword OCIDurationEnd ( OCIEnv *env, OCIError *err, const OCISvcCtx *svc, OCIDuration duration );

Parameters

env (IN/OUT)
Pass as a NULL pointer.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err and this function returns OCI_ERROR. Diagnostic information can be obtained by calling OCIErrorGet().

svc (IN)
OCI service context. This should be passed as NULL for cartridge services.

duration (IN)
A number to identify the user duration.

Comments
This function terminates a user duration. Temporary LOBs that are allocated for the user duration are freed.

See Also: "Temporary LOB Durations" on page 7-15

Related Functions
OCIDurationBegin()
**OCILobAppend()**

**Purpose**

Appends a LOB value at the end of another LOB as specified.

**Syntax**

```c
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 BLOBs or both be CLOBs). LOB buffering must not be enabled for either type of locator. This function does not accept a BFILE locator as the source or the destination.

It is not mandatory that you wrap this LOB operation inside the open or close calls. If you did not open the LOB 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 Functions

- OCILobTrim() (deprecated), OCILobTrim2(), OCILobWrite() (deprecated),
- OCILobWrite2(), OCILobCopy() (deprecated), OCILobCopy2(), OCIErrorGet(),
- OCILobWriteAppend() (deprecated), OCILobWriteAppend2()
OCILobArrayRead()

Purpose

Reads LOB data for multiple locators in one round-trip. This function can be used for LOBs of size greater than or less than 4 GB.

Syntax

```c
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 (CLOBs, NCLOBs) it is the number of characters from the beginning of the LOB; for binary LOBs or BFILES, it is the number of bytes. The first position is 1.

**bufp_arr (IN/OUT)**
An array of pointers to buffers into which the piece 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.

---

For BFILES, 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 CLOBs and NCLOBs, 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 CLOBs and NCLOBs, `char_amt_arr` returns the number of characters read in the buffer as well.

To read data in UTF-16 format, set the `csid` parameter to `OCI_UTF16ID`. If the `csid` parameter is set, it overrides the `NLS_LANG` environment variable.

**See Also:**

- "PL/SQL REF CURSORs and Nested Tables in OCI" on page 5-32 for additional information on Unicode format
- *Oracle Database SecureFiles and Large Objects Developer’s Guide* for a description of BFILEs
- The demonstration programs included with your Oracle Database installation for a code sample showing the use of LOB reads and writes.
- Appendix B, "OCI Demonstration Programs"
- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for general information about piecewise OCI operations

**Related Functions**

OCIErrorGet(), OCIlobWrite2(), OCIlobFileSetName(), OCIlobWriteAppend2(), OCIlobArrayWrite()
**OCILobArrayWrite()**

**Purpose**

Writes LOB data for multiple locators in one round-trip. This function can be used for LOBs of size greater than or less than 4 GB.

**Syntax**

```c
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 (CLOBs, NCLOBs), it is the number of characters from the beginning of the LOB; for BLOBs, 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.

csf rm (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.

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. BFILES 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 CLOBs and NCLOBs, `byte_amt_arr` and `char_amt_arr` return the data written by each piece in terms of number of bytes and number of characters respectively. For BLOBs, `byte_amt_arr` returns the number of bytes written by each piece, 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.

**See Also:**

- "PL/SQL REF CURSORs and Nested Tables in OCI" on page 5-32 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.
- Appendix B, "OCI Demonstration Programs"
- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for general information about piecewise OCI operations

**Related Functions**

- `OCIErrorGet()`, `OCILobRead2()`, `OCILobAppend()`, `OCILobCopy()` (deprecated), `OCILobCopy2()`, `OCILobWriteAppend2()`, `OCILobArrayRead()`
OCILobAssign()

Purpose

Assigns one LOB or BFILE locator to another.

Syntax

```c
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()** cannot be used for temporary LOBs; it generates an OCI_INVALID_HANDLE error. For temporary LOBs, use OCILobLocatorAssign().

For BFILES, only the locator that refers to the file is copied to the table. The operating system file itself is not copied.

It is an error to assign a BFILE locator to an internal LOB locator, and vice versa.

If the source locator is for an internal LOB that was enabled for buffering, and the source locator has been used to modify the LOB data through the LOB buffering subsystem, and the buffers have not been flushed since the write, then the source locator may not be assigned to the destination locator. This is because only one locator for each LOB 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:

```c
OCILobLocator *source_loc = (OCILobLocator *) 0;
OCILobLocator *dest_loc = (OCILobLocator *) 0;
```

**Example 17–1** shows how an application could allocate the source_loc locator.
**Example 17–1 Allocating a source_loc Source Locator**

```c
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 shown in Example 17–2.

**Example 17–2 Allocating a dest_loc Destination Locator**

```c
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 Functions**

- OCIErrorGet()
- OCILobIsEqual()
- OCILobLocatorAssign()
- OCILobLocatorIsInit()
- OCILobEnableBuffering()
**OCILobCharSetForm()**

**Purpose**

Gets the character set form of the LOB locator, if any.

**Syntax**

```c
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 BFILES. 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 Functions**

OCIErrorGet(), OCILobCharSetId(), OCILobLocatorIsInit()
**OCILobCharSetId()**

**Purpose**

Gets the LOB locator’s database character set ID of the LOB locator, if any.

**Syntax**

```c
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 Functions**

- OCIErrorGet(), OCILobCharSetForm(), OCILobLocatorIsInit()
OCILOBClose()

Purpose

Closes a previously opened LOB or BFILE.

Syntax

```c
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.

See Also: "Functions for Opening and Closing LOBs" on page 7-10

Related Functions

OCIErrorGet(), OCILobOpen(), OCILobIsOpen()
OCILOBCopy2()

Purpose
Copies all or a portion of a LOB value into another LOB value. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax
sword OCILobCopy2 ( OCISvcCtx *svchp,
OCIError *errhp,
OCILobLocator *dst_locp,
OCILobLocator *src_locp,
oraub8 amount,
dauser8 dst_offset,
dauser8 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 CLOBs or NCLOBs or the number of bytes for BLOBs 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 BLOBs) or spaces (for CLOBs) are written into the destination LOB from the end of the current data to the beginning of the newly written data from the source. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is an error to extend the destination LOB beyond the maximum length allowed (that is, 4 gigabytes) or to try to copy from a NULL LOB. 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 BLOBs or both be CLOBs). LOB buffering must not be enabled for either locator.

This function does not accept a BFILE locator as the source or the destination.

It is not mandatory that you wrap this LOB operation inside the open or close calls. If you did not open the LOB 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 Functions

OCIErrorGet(), OCILobRead2(), OCILobAppend(), OCILobWrite2(), OCILobWriteAppend2()
**OCILobCreateTemporary()**

**Purpose**

Creates a temporary LOB.

**Syntax**

```c
sword OCILobCreateTemporary(OCISvcCtx *svchp,
OCIErrror *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 `OCIErrrorGet()` 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 Oracle8i 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`
OCIlobCreateTemporary()

- OCI_DURATION_CALL

**Comments**

This function creates a temporary LOB and its corresponding index in the user's temporary tablespace.

When this function is complete, the locp parameter points to an empty temporary LOB whose length is zero.

The lifetime of the temporary LOB is determined by the duration parameter. At the end of its duration the temporary LOB is freed. An application can free a temporary LOB sooner with the OCILobFreeTemporary() call.

If the LOB is a BLOB, the csid and csfrm parameters are ignored.

**See Also:** "Temporary LOB Support" on page 7-14 for more information about temporary LOBs and their durations

**Related Functions**

OCILobFreeTemporary(), OCILobIsTemporary(), OCIDescriptorAlloc(), OCIErrorGet()
OCILobDisableBuffering()

Purpose

Disables LOB buffering for the input locator.

Syntax

sword OCILobDisableBuffering ( OCISvcCtx *svchp,
                         OCIError *errhp,
                         OCILobLocator *locp );

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle 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.

This function does not accept a BFILE locator.

Related Functions

OCILobEnableBuffering(), OCIErrorGet(), OCILobFlushBuffer()
OCILobEnableBuffering()

Purpose

Enables LOB buffering for the input locator.

Syntax

```c
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.

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 Functions

OCILobDisableBuffering(), OCIErrorGet(), OCILobWrite() (deprecated), OCILobWrite2(), OCILobRead() (deprecated), OCILobRead2(), OCILobFlushBuffer(), OCILobWriteAppend() (deprecated), OCILobWriteAppend2()
**OCILobErase2()**

**Purpose**

Erases a specified portion of the internal LOB data starting at a specified offset. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

**Syntax**

```c
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 CLOBs or NCLOBs, or bytes for BLOBs, to erase. On IN, the value signifies the number of characters or bytes to erase. On OUT, the value identifies the actual number of characters or bytes erased.

- **offset (IN)**
  Absolute offset in characters for CLOBs or NCLOBs, or bytes for BLOBs, from the beginning of the LOB value from which to start erasing data. Starts at 1.

**Comments**

The actual number of characters or bytes erased is returned. For BLOBs, erasing means that zero-byte fillers overwrite the existing LOB value. For CLOBs, erasing means that spaces overwrite the existing LOB value.

This function is valid only for internal LOBs; BFILEs 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 Functions

OCIErrorGet(), OCILobErase2(), OCILobRead2(), OCILobAppend(), OCILobCopy2(),
OCILobWrite2(), OCILobWriteAppend2()
OBILobFileClose()

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.

See Also: Oracle Database SecureFiles and Large Objects Developer’s Guide for a description of BFILES

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileIsOpen(), OCILobFileOpen(), OCILobOpen(), OCILobIsOpen()
OCILobFileCloseAll()

Purpose

Closes all open BFILEs on a given service context.

Syntax

sword OCILobFileCloseAll ( OCISvcCtx *svchp,
                           OCIError *errhp );

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle that you can pass to OCIErrorGet() for diagnostic information when there is an error.

Comments

Closes all open BFILEs on a given service context.

See Also: Oracle Database SecureFiles and Large Objects Developer's Guide for a description of BFILEs

Related Functions

OCILobFileClose(), OCIErrorGet(), OCILobFileExists(), OCILobFileIsOpen()
OCILobFileExists()

Purpose
Tests to see if the BFILE exists on the server’s operating system.

Syntax

```c
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.

See Also: Oracle Database SecureFiles and Large Objects
Developer’s Guide for a description of BFILES

Related Functions

OCIErrorGet(), OCILobFileClose(), OCILobFileCloseAll(), OCILobFileIsOpen(),
OCILobOpen(), OCILobIsOpen()
OCILobFileGetName()

Purpose

Gets the BFILE locator's directory object and file name.

Syntax

```c
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.

See Also: *Oracle Database SecureFiles and Large Objects Developer’s Guide* for a description of BFILEs
Related Functions

OCILobFileSetName(), OCIErrorGet()
OCILobFileIsOpen()

Purpose

Tests to see if the BFILE is open.

Syntax

```c
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.

See Also: Oracle Database SecureFiles and Large Objects Developer’s Guide for a description of BFILES

Related Functions

- OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileClose(), OCILobFileOpen(), OCILobOpen(), OCILobIsOpen()
OCILobFileOpen()

Purpose

Opens a BFILE on the file system of the server for read-only access.

Syntax

```c
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. BFILES 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.

**See Also:** Oracle Database SecureFiles and Large Objects Developer’s Guide for a description of BFILES

Related Functions

- OCIErrorGet()
- OCILobClose()
- OCILobFileCloseAll()
- OCILobFileExists()
- OCILobFileClose()
- OCILobFileIsOpen()
- OCILobOpen()
- OCILobIsOpen()
OCILobFileSetName()

Purpose

Sets the directory object and file name in the BFILE locator.

Syntax

```c
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.

See Also: Oracle Database SecureFiles and Large Objects Developer’s Guide for a description of BFILES

Related Functions

OCILobFileGetName(), OCIErrorGet()
**OCILobFlushBuffer()**

**Purpose**
Flushes or write s all buffers for this LOB to the server.

**Syntax**
```c
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.

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 Functions**
- OCILobEnableBuffering(), OCIErrorGet(), OCILobWrite() (deprecated),
- OCILobWrite2(), OCILobRead() (deprecated), OCILobRead2(),
- OCILobDisableBuffering(), OCILobWriteAppend() (deprecated),
- OCILobWriteAppend2()
OCILobFreeTemporary()

Purpose

Frees a temporary LOB.

Syntax

```c
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.

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 functions

- OCILobCreateTemporary()
- OCILobIsTemporary()
- OCIErrorGet()
**OCILobGetChunkSize()**

**Purpose**

Gets the chunk size of a LOB.

**Syntax**

```c
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 BLOBs, CLOBs, and NCLOBs.

  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.

**See Also:** "Improving LOB Read/Write Performance" on page 7-8

**Related Functions**

- OCIErrorGet(), OCILobGetStorageLimit(), OCILobRead() (deprecated),
- OCILobRead2(), OCILobAppend(), OCILobCopy() (deprecated), OCILobCopy2(),
- OCILobWrite() (deprecated), OCILobWrite2(), OCILobWriteAppend() (deprecated),
- OCILobWriteAppend2()
OCILobGetContentType()

**Purpose**

Gets the user-specified content type string for the data in a SecureFile, if set.

**Syntax**

```c
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 \(\geq\) 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 Content Type string is defined as:

```
#define OCI_LOB_CONTENTTYPE_MAXSIZE 128
```

The Content Type is ASCII (that is, 1-byte/7-bit UTF8).
Related Functions

OCILOBSetContentType()
**OCILobGetLength2()**

**Purpose**

Gets the length of a LOB. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

**Syntax**

```c
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 BFILES, the locator can be set by OCILobFileSetName(), by a SELECT statement, or by OCIObjectPin().

- **lenp (OUT)**
  On output, it is the length of the LOB if the LOB is not NULL. For character LOBs, it is the number of characters; for binary LOBs and BFILES, it is the number of bytes in the LOB.

**Comments**

Gets the length of a LOB. If the LOB is NULL, the length is undefined. The length of a 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 CLOBs and NCLOBs, and in bytes for BLOBs and BFILES.

---

**Note:** Any zero-byte or space fillers in the LOB written by previous calls to OCILobErase2() or OCILobWrite2() are also included in the length count.

---

**Related Functions**

OCIErrorGet(), OCILobFileSetName(), OCILobGetLength2(), OCILobRead2(),
OCILobWrite2(), OCILobCopy2(), OCILobAppend(), OCILobLoadFromFile2(),
OCILobWriteAppend2()
OCILobGetOptions()

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 Functions

OCILobSetOptions()
**OCILOBGetStorageLimit()**

**Purpose**

Gets the maximum length of an internal LOB (BLOB, CLOB, or NCLOB) in bytes.

**Syntax**

```c
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.

**See Also:** "Using LOBs of Size Greater than 4 GB" on page 7-4

**Related Functions**

- OCILOBGetChunkSize()
OCILobIsEqual()

Purpose

Compares two LOB or BFILE locators for equality.

Syntax

```c
sword OCILobIsEqual ( OCIEnv                  *envhp,
                       const OCILobLocator     *x,
                       const OCILobLocator     *y,
                       boolean                 *is_equal );
```

Parameters

- **envhp (IN)**
  The OCI environment handle.

- **x (IN)**
  LOB locator to compare.

- **y (IN)**
  LOB locator to compare.

- **is_equal (OUT)**
  TRUE, if the LOB locators are equal; FALSE if they are not.

Comments

Compares the given LOB or BFILE locators for equality. Two LOB or BFILE locators are equal if and only if they both refer to the same LOB or BFILE value.

Two NULL locators are considered not equal by this function.

Related Functions

- OCILobAssign()
- OCILobLocatorIsInit()
OCILobIsOpen()

Purpose
Tests whether a LOB or BFILE is open.

Syntax
sword OCILobIsOpen ( OCISvcCtx *svchp, 
OCIError *errhp, 
OCILobLocator *locp, 
boolean *flag );

Parameters
svchp (IN)
The service context handle.
errhp (IN/OUT)
An error handle 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 BFILES.

For internal LOBs
Openness is associated with the LOB, not with the locator. If locator1 opened the LOB, then locator2 also sees the LOB as open.

For internal LOBs, this call requires a server round-trip because it checks the state on the server to see if the LOB is open. For external LOBs (BFILES), 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.

See Also: "Functions for Opening and Closing LOBs" on page 7-10
Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(),
OCILobFileClose(), OCILobFileIsOpen(), OCILobFileOpen(), OCILobOpen()
OCILobIsTemporary()

Purpose

Tests if a locator points to a temporary LOB

Syntax

```c
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 Functions

- OCILobCreateTemporary(), OCILobFreeTemporary()
OCILobLoadFromFile2()

Purpose

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. You can also use this function for LOBs smaller than 4 GB.

Syntax

```c
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.
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 BLOBs) or spaces (for CLOBs) are written into the destination LOB from the end of the data to the beginning of the newly written data from the source. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB.

It is an error to extend the destination LOB beyond the maximum length allowed (4 gigabytes) (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 Functions**

OCIErrorGet(), OCILobAppend(), OCILobWrite2(), OCILobTrim2(), OCILobCopy2(), OCILobGetLength2(), OCILobLoadFromFile2(), OCILobWriteAppend2()
**OCILOBLocatorAssign()**

**Purpose**

Assigns one LOB or BFILE locator to another.

**Syntax**

```c
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 BFILES, only the locator that refers to the operating system file is copied to the table; the operating system file is not copied.

Note that this call is similar to **OCILOBAssign()**, but **OCILOBLocatorAssign()** takes an OCI service handle pointer instead of an OCI environment handle pointer. Also, **OCILOBLocatorAssign()** can be used for temporary LOBs, 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 Functions**

`OCIErrorGet()`, `OCILobAssign()`, `OCILobisEqual()`, `OCILobLocatorIsInit()`
OCILobLocatorIsInit()

**Purpose**
Tests to see if a given LOB or BFILE locator is initialized.

**Syntax**

```c
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 Locator Attributes" on page A-39

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 Functions**

`OCIErrorGet()`, `OCILobIsEqual()`
LOB Functions

OCILobOpen()

Purpose

Opens a LOB, internal or external, in the indicated mode.

Syntax

```c
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 Oracle8i or later, valid modes for LOBs are OCI_LOB_READONLY and OCI_LOB_READWRITE. Note that OCI_FILE_READONLY exists as input to OCILobFileOpen(). OCI_FILE_READONLY can be used with OCILobOpen() if the input locator is for a BFILE.

Comments

It is an error to open the same LOB twice. BFILES cannot be opened in read/write mode. 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 (BFILES), open requires a round-trip because the actual operating system file on the server side is being opened.

It is not necessary to open a LOB 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.

See Also: "Functions for Opening and Closing LOBs" on page 7-10

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileClose(), OCILobFileIsOpen(), OCILobFileOpen(), OCILobIsOpen()
OCILobRead2()

Purpose
Reads a portion of a LOB or BFILE, as specified by the call, into a buffer. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax
```c
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 (CLOBs, NCLOBs), it is the number of characters from the beginning of the LOB; for
binary LOBs or BFILEs, it is the number of bytes. The first position is 1.
If you use streaming (by polling or a callback), 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 CLOBs and
for NCLOBs (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 BFILES, 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 CLOBs and NCLOBs, `char_amtp` gives the number of characters read in the current piece.

- 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 CLOBs and NCLOBs, `char_amtp` returns the number of characters read in the buffer as well.

To read data in UTF-16 format, set the `csid` parameter to `OCI_UTF16ID`. If the `csid` parameter is set, it overrides the `NLS_LANG` environment variable.
See Also:

- "PL/SQL REF CURSORs and Nested Tables in OCI" on page 5-32 for additional information about Unicode format
- Oracle Database SecureFiles and Large Objects Developer’s Guide for a description of BFILES
- The demonstration programs included with your Oracle Database installation for a code sample showing the use of LOB reads and writes.
- Appendix B, "OCI Demonstration Programs"
- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for general information about piecewise OCI operations
- "Polling Mode Operations in OCI" on page 2-27

Related Functions

OCIErrorGet(), OCILobWrite2(), OCILobFileName(), OCILobWriteAppend2()
OCILobSetContentType()

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.

csvchp (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.
OCILobSetContentType()

Related Functions

OCILobGetContentType()
**OCILobSetOptions()**

**Purpose**

Enables option settings for a SecureFile LOB.

**Syntax**

```c
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`
  - Deduplication - `OCI_LOB_OPT_DEDUPLICATE`

- **optionsp (IN)**
  The possible settings are:
  - `OCI_LOB_OPT_COMPRESS_OFF`
  - `OCI_LOB_OPT_COMPRESS_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 Functions**

- `OCILobGetOptions()`
OCILobTrim2()

Purpose

Truncates the LOB value to a shorter length. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```c
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 BFILES, it is the number of bytes in the LOB.

Comments

This function trims the LOB data to a specified shorter length. The function returns an error if `newlen` is greater than the current LOB length. This function is valid only for internal LOBs. BFILES 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 Functions

- OCIErrorGet(), OCILobRead2(), OCILobAppend(), OCILobCopy2(), OCILobErase2(), OCILobTrim2(), OCILobWrite2(), OCILobWriteAppend2()
OCILobWrite2()

Purpose

Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```c
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 CLOBs and NCLOBs 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.

cctxp (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:

cctxp (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.

csfm (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. BFILES 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 CLOBs and NCLOBs, `byte_amtp` and `char_amtp` return the data written by each piece in terms of number of bytes and number of characters respectively. For BLOBs, `byte_amtp` returns the number of bytes written by each piece, 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.

**See Also:**

- "PL/SQL REF CURSORs and Nested Tables in OCI" on page 5-32 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.
- Appendix B, "OCI Demonstration Programs"
- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for general information about piecewise OCI operations
- "Polling Mode Operations in OCI" on page 2-27

**Related Functions**

`OCIErrorGet()`, `OCILobRead2()`, `OCILobAppend()`, `OCILobCopy2()`, `OCILobWriteAppend2()`
OCILobWriteAppend2()

Purpose

Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```c
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 *leng,
    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.

**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**.

**ctxxp (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:

- **ctxxp (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 zero amount and `OCI_FIRST_PIECE` is given as input, then polling mode is assumed and data is written until you specify `OCI_LAST_PIECE`. For CLOBs and NCLOBs, `byte_amtp` and `char_amtp` return the data written by each piece in terms of number of bytes and number of characters respectively. For BLOBs, `byte_amtp` returns the number of bytes written by each piece 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.

See Also: "Improving LOB Read/Write Performance" on page 7-8

Related Functions

`OCIErrorGet()`, `OCILobRead2()`, `OCILobAppend()`, `OCILobCopy2()`, `OCILobWrite2()`
Table 17–3 lists the Streams Advanced Queuing and publish-subscribe functions that are described in this section. Use functions that end in "2" for all new applications.

See Also: "OCI Demonstration Programs" on page B-1 for Streams Advanced Queuing programs

<table>
<thead>
<tr>
<th>Table 17–3 Advanced Queuing and Publish-Subscribe Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
</tr>
<tr>
<td>&quot;OCIAQDeq()&quot; on page 17-89</td>
</tr>
<tr>
<td>&quot;OCIAQDeqArray()&quot; on page 17-91</td>
</tr>
<tr>
<td>&quot;OCIAQEnq()&quot; on page 17-93</td>
</tr>
<tr>
<td>&quot;OCIAQEnqArray()&quot; on page 17-95</td>
</tr>
<tr>
<td>&quot;OCIAQListen2()&quot; on page 17-97</td>
</tr>
<tr>
<td>&quot;OCISubscriptionDisable()&quot; on page 17-99</td>
</tr>
<tr>
<td>&quot;OCISubscriptionEnable()&quot; on page 17-100</td>
</tr>
<tr>
<td>&quot;OCISubscriptionPost()&quot; on page 17-101</td>
</tr>
<tr>
<td>&quot;OCISubscriptionRegister()&quot; on page 17-103</td>
</tr>
<tr>
<td>&quot;OCISubscriptionUnRegister()&quot; on page 17-105</td>
</tr>
</tbody>
</table>
OCIAQDeq()

Purpose

Performs a dequeue operation using Streams Advanced Queuing with OCI.

Syntax

```c
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 10g 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" on page 11-9.

**payload_ind (IN/OUT)**
A pointer to a pointer to the program variable buffer containing the parallel indicator structure for the object type.

The memory allocation rules for payload_ind are the same as those for payload.

**msgid (OUT)**
The message ID.

**flags (IN)**
Not currently used; pass as OCI_DEFAULT.

**Comments**

Users must have the AQ_USER_ROLE or privileges to execute the DBMS_AQ package 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 Streams Advanced Queuing" on page 9-47
- Oracle Streams Advanced Queuing User’s Guide

**Related Functions**

OCIAQEnq(), OCIAQListen2(), OCIInitialize() (deprecated), OCIEnvCreate(), OCIEnvNlsCreate()
OCIAQDeqArray()

Purpose
Dequeues an array of messages from a queue. 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 10g 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.
**OCIAQDeqArray()**

**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.

**See Also:**

- "OCI and Streams Advanced Queuing" on page 9-47
- Oracle Streams Advanced Queuing User’s Guide

**Related Functions**

OCIAQDeq(), OCIAQEnqArray(), OCIAQListen2(), OCIInitialize() (deprecated), OCIEnvCreate(), OCIEnvNlsCreate()
OCIAQEnq()

Purpose

Performs an enqueue operation using Streams Advanced Queuing.

Syntax

```c
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 10g 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" on page 11-9 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 Streams Advanced Queuing" on page 9-47 for more information about OCI and Advanced Queuing
■ Oracle Streams Advanced Queuing User’s Guide

To obtain a TDO for the payload, use OCITypeByName(), or OCITypeByRef().

Related Functions

OCIAQDeq(), OCIAQListen2(), OCIInitialize() (deprecated), OCIEnvCreate(), OCIEnvNlsCreate()
OCIAQEnqArray()

**Purpose**
Enqueues an array of messages to a queue. The array of messages is enqueued with the same options and has the same payload column TDO.

**Syntax**
```c
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 10g 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.

cctxp (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 Functions
OCIAQEnq(), OCIAQDeqArray(), OCIAQListen2(), OCIInitialize() (deprecated), OCIEnvCreate(), OCIEnvNlsCreate()
OCIAQListen2()

Purpose

Listens on one or more queues on behalf of a list of agents. Supports buffered messaging and persistent queues. Introduced in Oracle Database 10g 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.

erhpp (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 Functions  
OCIAQEnq(), OCIAQDeq(), OCISvcCtxToLda(), OCISubscriptionEnable(), OCISubscriptionPost(), OCISubscriptionRegister(), OCISubscriptionUnRegister()
OCISubscriptionDisable()

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.

- `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.

See Also: "Subscription Handle Attributes" on page A-51

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 Functions

- `OCIAQListen2()`, `OCISubscriptionEnable()`, `OCISubscriptionPost()`, `OCISubscriptionRegister()`, `OCISubscriptionUnRegister()`
OCISubscriptionEnable()

Purpose

Enables a subscription registration that has been disabled. This turns on all notifications.

Syntax

```c
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" on page A-51

- `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 Functions

- `OCIAQListen2()`, `OCISvcCtxToLda()`, `OCISubscriptionPost()`, `OCISubscriptionRegister()`, `OCISubscriptionUnRegister()`
OCISubscriptionPost()

Purpose

Posts to a subscription that allows all clients who are registered for the subscription to get notifications.

Syntax

```c
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.

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 Functions

OCIAQListen2(), OCISvcCtxToLda(), OCISubscriptionEnable(),
OCISubscriptionRegister(), OCISubscriptionUnRegister()
OCISubscriptionRegister()

Purpose

Registers a callback for message notification.

Syntax

```c
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" on page A-51 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_Subscription_Register()

- **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 Functions**

`OCIAQListen2()`, `OCISvcCtxToLda()`, `OCISubscriptionEnable()`, `OCISubscriptionPost()`, `OCISubscriptionUnRegister()`
OCISubscriptionUnRegister()

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.

- **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_LDAONLY` - 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 Functions

- `OCIAQListen2()`, `OCISvcCtxToLda()`, `OCISubscriptionEnable()`, `OCISubscriptionPost()`, `OCISubscriptionRegister()`
Table 17–4 lists the direct path loading functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIDirPathAbort()&quot; on page 17-107</td>
<td>Terminate a direct path operation</td>
</tr>
<tr>
<td>&quot;OCIDirPathColArrayEntryGet()&quot; on page 17-108</td>
<td>Get a specified entry in a column array</td>
</tr>
<tr>
<td>&quot;OCIDirPathColArrayEntrySet()&quot; on page 17-109</td>
<td>Set a specified entry in a column array to a specific value</td>
</tr>
<tr>
<td>&quot;OCIDirPathColArrayReset()&quot; on page 17-111</td>
<td>Reset the row array state</td>
</tr>
<tr>
<td>&quot;OCIDirPathColArrayRowGet()&quot; on page 17-112</td>
<td>Get the base row pointers for a specified row number</td>
</tr>
<tr>
<td>&quot;OCIDirPathColArrayToStream()&quot; on page 17-113</td>
<td>Convert from a column array to a direct path stream format</td>
</tr>
<tr>
<td>&quot;OCIDirPathDataSave()&quot; on page 17-115</td>
<td>Do a data savepoint, or commit the loaded data and finish the load operation</td>
</tr>
<tr>
<td>&quot;OCIDirPathFinish()&quot; on page 17-116</td>
<td>Finish and commit the loaded data</td>
</tr>
<tr>
<td>&quot;OCIDirPathFlushRow()&quot; on page 17-117</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>&quot;OCIDirPathLoadStream()&quot; on page 17-118</td>
<td>Load the data converted to direct path stream format</td>
</tr>
<tr>
<td>&quot;OCIDirPathPrepare()&quot; on page 17-119</td>
<td>Prepare direct path interface to convert or load rows</td>
</tr>
<tr>
<td>&quot;OCIDirPathStreamReset()&quot; on page 17-120</td>
<td>Reset the direct path stream state</td>
</tr>
</tbody>
</table>
**OCIDirPathAbort()**

**Purpose**
Terminates a direct path operation.

**Syntax**
```c
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 Functions**
- `OCIDirPathFinish()`, `OCIDirPathPrepare()`, `OCIDirPathLoadStream()`, `OCIDirPathStreamReset()`, `OCIDirPathDataSave()`
**OCIDirPathColArrayEntryGet()**

**Purpose**

Gets a specified entry in a column array.

**Syntax**

```c
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 Functions**

- OCIDirPathColArrayEntrySet()
- OCIDirPathColArrayRowGet()
- OCIDirPathColArrayReset()
- OCIDirPathColArrayToStream()
**OCIDirPathColArrayEntrySet()**

**Purpose**
Sets a specified entry in a column array to the supplied values.

**Syntax**
```
sword OCIDirPathColArrayEntrySet ( OCIDirPathColArray   *dpca,
                                 OCIError             *errhp,
                                 ub4                  rownum,
                                 ub2                  colIdx,
                                 ub1                  *cvalp,
                                 ub4                  clen,
                                 ub1                  cflg );
```

**Parameters**
- **dpca (IN/OUT)**
  Direct path column array handle.
- **errhp (IN)**
  An error handle 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 Functions

- OCIDirPathColArrayRowGet()
- OCIDirPathColArrayRowGet()
- OCIDirPathColArrayReset()
- OCIDirPathColArrayToStream()
OCIDirPathColArrayReset()

Purpose

Resets the column array state.

Syntax

```c
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 Functions

- OCIDirPathColArrayEntryGet(), OCIDirPathColArrayEntrySet(),
- OCIDirPathColArrayRowGet(), OCIDirPathColArrayToStream()
OCIDirPathColArrayRowGet()

Purpose

Gets the column array row pointers for a given row number.

Syntax

```c
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 Functions

- OCIDirPathColArrayRowGet()
- OCIDirPathColArrayEntrySet()
- OCIDirPathColArrayReset()
- OCIDirPathColArrayToStream()
**OCIDirPathColArrayToStream()**

**Purpose**

Converts from column array format to a direct path stream format.

**Syntax**

```c
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 Functions

OCIDirPathColArrayEntryGet(), OCIDirPathColArrayEntrySet(), OCIDirPathColArrayRowGet(), OCIDirPathColArrayReset()
OCIDirPathDataSave()

Purpose
Depending on the action requested, does a data savepoint, or commits the loaded data and finishes the direct path load operation.

Syntax
sword OCIDirPathDataSave ( OCIDirPathCtx          *dpctx,
                         OCIError               *errhp,
                         ub4                    action  );

Parameters

dpctx (IN)
Direct path context handle for the object loaded.

errhp (IN/OUT)
An error handle 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 Functions
OCIDirPathAbort(), OCIDirPathFinish(), OCIDirPathPrepare(),
OCIDirPathStreamReset()
OCIDirPathFinish()

Purpose
Finishes the direct path load operation.

Syntax
sword OCIDirPathFinish (   OCIDirPathCtx          *dpctx,
                          OCIError               *errhp );

Parameters

dpctx (IN)
Direct path context handle for the object loaded.

errhp (IN/OUT)
An error handle 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 Functions
OCIDirPathAbort(), OCIDirPathDataSave(), OCIDirPathPrepare(),
OCIDirPathStreamReset()
**OCIDirPathFlushRow()**

**Purpose**
Flushes a partially loaded row from the server. This function is deprecated.

**Syntax**
```c
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 Functions**
- OCIDirPathAbort(), OCIDirPathFinish(), OCIDirPathPrepare(),
  OCIDirPathLoadStream()
**OCIDirPathLoadStream()**

**Purpose**

Loads the data converted to direct path stream format.

**Syntax**

```c
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 Functions**

- OCIDirPathAbort(), OCIDirPathDataSave(), OCIDirPathFinish(),
- OCIDirPathPrepare(), OCIDirPathStreamReset()
OCIDirPathPrepare()

Purpose
Prepares the direct path load interface before any rows can be converted or loaded.

Syntax
sword OCIDirPathPrepare (   OCIDirPathCtx          *dpctx,
OCISvcCtx              *svchp,
OCIError               *errhp );

Parameters

dpctx (IN)
Direct path context handle for the object loaded.

svchp (IN)
Service context.

errhp (IN/OUT)
An error handle 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 Functions
OCIDirPathAbort(), OCIDirPathDataSave(), OCIDirPathFinish(), OCIDirPathStreamReset()
OCIPathStreamReset()

Purpose

Resets the direct path stream state.

Syntax

sword OCIDirPathStreamReset ( OCIPathStream       *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 Functions

  OCIDirPathAbort(), OCIDirPathDataSave(), OCIDirPathFinish(), OCIDirPathPrepare()
Thread Management Functions

Table 17–5 lists the thread management functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIThreadClose()</td>
<td>Close a thread handle</td>
</tr>
<tr>
<td>OCIThreadCreate()</td>
<td>Create a new thread</td>
</tr>
<tr>
<td>OCIThreadHandleGet()</td>
<td>Retrieve the OCIThreadHandle of the thread in which it is called</td>
</tr>
<tr>
<td>OCIThreadHndDestroy()</td>
<td>Destroy and deallocate the thread handle</td>
</tr>
<tr>
<td>OCIThreadHndInit()</td>
<td>Allocate and initialize the thread handle</td>
</tr>
<tr>
<td>OCIThreadIdDestroy()</td>
<td>Destroy and deallocate a thread ID</td>
</tr>
<tr>
<td>OCIThreadIdGet()</td>
<td>Retrieve the OCIThreadId of the thread in which it is called</td>
</tr>
<tr>
<td>OCIThreadIdInit()</td>
<td>Allocate and initialize the thread ID</td>
</tr>
<tr>
<td>OCIThreadIdNull()</td>
<td>Determine whether a given OCIThreadId is the NULL thread ID</td>
</tr>
<tr>
<td>OCIThreadIdSame()</td>
<td>Determine whether two OCIThreadIds represent the same thread</td>
</tr>
<tr>
<td>OCIThreadIdSet()</td>
<td>Set one OCIThreadId to another</td>
</tr>
<tr>
<td>OCIThreadIdSetNull()</td>
<td>Set the NULL thread ID to a given OCIThreadId</td>
</tr>
<tr>
<td>OCIThreadInit()</td>
<td>Initialize OCIThread context</td>
</tr>
<tr>
<td>OCIThreadIsMulti()</td>
<td>Tell the caller whether the application is running in a multithreaded environment or a single-threaded environment</td>
</tr>
<tr>
<td>OCIThreadJoin()</td>
<td>Allow the calling thread to join with another thread</td>
</tr>
<tr>
<td>OCIThreadKeyDestroy()</td>
<td>Destroy and deallocate the key pointed to by key</td>
</tr>
<tr>
<td>OCIThreadKeyGet()</td>
<td>Get the calling thread’s current value for a key</td>
</tr>
<tr>
<td>OCIThreadKeyIdInit()</td>
<td>Create a key</td>
</tr>
<tr>
<td>OCIThreadKeyIdSet()</td>
<td>Set the calling thread’s value for a key</td>
</tr>
<tr>
<td>OCIThreadMutexAcquire()</td>
<td>Acquire a mutex for the thread in which it is called</td>
</tr>
<tr>
<td>OCIThreadMutexDestroy()</td>
<td>Destroy and deallocate a mutex</td>
</tr>
<tr>
<td>OCIThreadMutexInit()</td>
<td>Allocate and initialize a mutex</td>
</tr>
<tr>
<td>OCIThreadMutexRelease()</td>
<td>Release a mutex</td>
</tr>
<tr>
<td>OCIThreadProcessInit()</td>
<td>Perform OCIThread process initialization</td>
</tr>
<tr>
<td>OCIThreadTerm()</td>
<td>Release the OCIThread context</td>
</tr>
</tbody>
</table>
OCIThreadClose()

Purpose

Closes a thread handle.

Syntax

```c
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 Functions

- `OCIThreadCreate()`
**OCIThreadCreate()**

**Purpose**

Creates a new thread.

**Syntax**

```c
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 Functions**

- `OCIThreadClose()`, `OCIThreadIdInit()`, `OCIThreadHndInit()`
OCIThreadHandleGet()

Purpose

Retrieves the OCIThreadHandle of the thread in which it is called.

Syntax

```c
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 Functions

**OCIThreadHndDestroy(), OCIThreadHndInit(), OCIThreadClose**
OCIThreadHndDestroy()

**Purpose**
Destroys and deallocates the thread handle.

**Syntax**
```c
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 Functions**
- `OCIThreadHandleGet()`, `OCIThreadHndInit()`
OCIThreadHndInit()

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 Functions

OCIThreadHandleGet(), OCIThreadHndDestroy()
**OCIThreadOIdDestroy()**

**Purpose**
Destroys and deallocates a thread ID.

**Syntax**
```c
sword OCIThreadOIdDestroy (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 Functions**

- **OCIThreadOIdGet()**, **OCIThreadOIdInit()**, **OCIThreadOIdNull()**, **OCIThreadOIdSame()**, **OCIThreadOIdSet()**, **OCIThreadOIdSetNull()**
**OCIThreadThreadIdGet()**

**Purpose**

Retrieves the OCIThreadId of the thread in which it is called.

**Syntax**

```c
sword OCIThreadThreadIdGet ( 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, `OCIThreadThreadIdGet()` 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 Functions**

- `OCIThreadThreadIdDestroy()`, `OCIThreadThreadIdInit()`, `OCIThreadThreadIdNull()`, `OCIThreadThreadIdSame()`, `OCIThreadThreadIdSet()`, `OCIThreadThreadIdSetNull()`
OCIThreadldInit()

Purpose

Allocate and initialize the thread ID tid.

Syntax

```c
sword OCIThreadldInit ( 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 Functions

OCIThreadldDestroy(), OCIThreadldGet(), OCIThreadldNull(), OCIThreadldSame(), OCIThreadldSet(), OCIThreadldSetNull()
OCIThreadIdNull()

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 OCIErrGet().

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 Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdSame(), OCIThreadIdSet(), OCIThreadIdSetNull()
OCIThreadIdsame()

Purpose
Determines whether two OCIThreadIds represent the same thread.

Syntax
sword OCI_ThreadIdsame ( 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 Functions
OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdNull(), OCIThreadIdSet(), OCIThreadIdSetNull()
**OCIThreadIdSet()**

**Purpose**
Sets one OCIThreadId to another.

**Syntax**
```c
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 Functions**
- OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdNull(), OCIThreadIdSame(), OCIThreadIdSetNull()
OCIThreadIdSetNull()

Purpose

Sets the NULL thread ID to a given OCIThreadId.

Syntax

```c
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 Functions

- `OCIThreadIdDestroy()`, `OCIThreadIdGet()`, `OCIThreadIdInit()`, `OCIThreadIdNull()`, `OCIThreadIdSame()`, `OCIThreadIdSet()`
OCIThreadInit()

Purpose
Initializes the OCIThread context.

Syntax
sword OCIThreadInit ( void *hndl,
OCIError *err );

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

e (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().

Related Functions

OCIThreadTerm()
OCIThreadIsMulti()

Purpose
Tells the caller whether the application is running in a multithreaded environment or a single-threaded environment.

Syntax
boolean OCIThreadIsMulti ( );

Returns
TRUE if the environment is multithreaded.
FALSE if the environment is single-threaded.

Related Functions
OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdNull(), OCIThreadIdSame(), OCIThreadIdSet()
OCIThreadJoin()

Purpose

Allows the calling thread to join with another thread.

Syntax

```c
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 Functions

- `OCIThreadIdDestroy()`, `OCIThreadIdGet()`, `OCIThreadIdInit()`, `OCIThreadIdNull()`, `OCIThreadIdSame()`, `OCIThreadIdSet()`
OCIThreadKeyDestroy()

Purpose

Destroys and deallocates the key pointed to by key.

Syntax

```c
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 Functions

OCIThreadKeyGet(), OCIThreadKeyInit(), OCIThreadKeySet()
OCIThreadKeyGet()

Purpose

Gets the calling thread’s current value for a key.

Syntax

```c
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 Functions

OCIThreadKeyDestroy(), OCIThreadKeyInit(), OCIThreadKeySet()
OCIThreadKeyInit()

Purpose

Creates a key.

Syntax

```c
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 Functions

`OCIThreadKeyDestroy()`, `OCIThreadKeyGet()`, `OCIThreadKeySet()`
OCIThreadKeySet()

Purpose

Sets the calling thread’s value for a key.

Syntax

```c
sword OCIThreadKeySet ( void           *hndl,
                     OCIError       *err,
                     OCIThreadKey   *key,
                     void           *value );
```

Parameters

- **hdl (IN/OUT)**
The OCI environment or user session handle.

- **err (IN/OUT)**
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in err and diagnostic information can be obtained by calling OCIErrorGet().

- **key (IN/OUT)**
The key.

- **value (IN)**
The thread-specific value to set in the key.

Comments

It is illegal to use this function on a key that has not been created using OCIThreadKeyInit().

Related Functions

OCIThreadKeyDestroy(), OCIThreadKeyGet(), OCIThreadKeyInit()
OCIThreadMutexAcquire()

Purpose

Acquires a mutex for the thread in which it is called.

Syntax

```c
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 Functions

- `OCIThreadMutexDestroy()`, `OCIThreadMutexInit()`, `OCIThreadMutexRelease()`
OCIThreadMutexDestroy()

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 Functions
OCIThreadMutexAcquire(), OCIThreadMutexInit(), OCIThreadMutexRelease()
OCIThreadMutexInit()

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 Functions
OCIThreadMutexDestroy(), OCIThreadMutexAcquire(), OCIThreadMutexRelease()
OCIThreadMutexRelease()

Purpose

Releases a mutex.

Syntax

```c
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 Functions

OCIThreadMutexDestroy(), OCIThreadMutexInit(), OCIThreadMutexAcquire()
Thread Management Functions

OCIThreadProcessInit()

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 Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdNull(), OCIThreadIdSame(), OCIThreadIdSet()
OCIThreadTerm()

Purpose

Releases the OCIThread context.

Syntax

```c
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 Functions

**OCIThreadInit()**
Table 17–6 lists the transaction functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCITransCommit() on page 17-148</td>
<td>Commit a transaction on a service context</td>
</tr>
<tr>
<td>OCITransDetach() on page 17-151</td>
<td>Detach a transaction from a service context</td>
</tr>
<tr>
<td>OCITransForget() on page 17-152</td>
<td>Forget a prepared global transaction</td>
</tr>
<tr>
<td>OCITransMultiPrepare() on page 17-153</td>
<td>Prepare a transaction with multiple branches in a single cell</td>
</tr>
<tr>
<td>OCITransPrepare() on page 17-154</td>
<td>Prepare a global transaction for commit</td>
</tr>
<tr>
<td>OCITransRollback() on page 17-155</td>
<td>Roll back a transaction</td>
</tr>
<tr>
<td>OCITransStart() on page 17-156</td>
<td>Start a transaction on a service context</td>
</tr>
</tbody>
</table>
OCITransCommit()

Purpose

Commits the transaction associated with a specified service context.

Syntax

```c
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_WRITENOWAIT** - 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. 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.

Caution: 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

Example 17–3 demonstrates the use of a simple local transaction, as described in "Simple Local Transactions" on page 8-2.

Example 17–3  Using OCITransCommit() in a Simple Local Transaction

```c
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);
```
OCI{Logon(envhp, errhp, &svchp, (text *)"HR", strlen("HR"),
   (text *)"HR", strlen("HR"), 0, 0);

   /* update hr.employees employee_id=7902, increment salary */
   sprintf((char *)sqlstmt, "UPDATE EMPLOYEES SET SALARY = SALARY + 1 \ 
   WHERE EMPLOYEE_ID = 7902");

   OCIStmtPrepare(stmthp, errhp, sqlstmt, strlen((char *)sqlstmt),
      OCI_NTV_SYNTAX, 0);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);
   OCITransCommit(svchp, errhp, (ub4) 0);

   /* update hr.employees employee_id=7902, increment salary again, but rollback */
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);
   OCITransRollback(svchp, errhp, (ub4) 0);
}

Related Functions
OCI{TransRollback()
OCI.TransDetach()

Purpose

Detaches a global transaction.

Syntax

```c
sword OCI.TransDetach ( OCISvcCtx *svchp,
       OCIError    *errhp,
       ub4          flags );
```

Parameters

- `svchp (IN)`
  The service context handle.

- `errhp (IN)`
  An error handle that you can pass to `OCI.ErrorGet()` 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 `OCI.TransStart()`, specifying a flags value of `OCI_TRANS_RESUME`.

When a transaction is detached, the value that was specified in the `timeout` parameter of `OCI.TransStart()` 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 `OCI.TransDetach()`, see the Examples section of `OCI.TransStart()`.

Related Functions

- `OCI.TransStart()`
OCITransForget()

Purpose
Causes the server to forget a heuristically completed global transaction.

Syntax
sword OCITransForget ( OCISvcCtx *svchp,
                      OCIError *errhp,
                      ub4 flags );

Parameters

  svchp (IN)
The service context handle in which the transaction resides.

  errhp (IN)
An error handle 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 Functions
OCITransCommit(), OCITransRollback()
OCITransMultiPrepare()

Purpose
Prepares a transaction with multiple branches in a single call.

Syntax
sword OCITransMultiPrepare ( OCISvcCtx   *svchp,
                           ub4         numBranches,
                           OCITrans    **txns,
                           OCIError    **errhp);

Parameters

srvchp (IN)
The service context handle.

numBranches (IN)
The number of branches expected. It is also the array size for the next two parameters.

txns (IN)
The array of transaction handles for the branches to prepare. They should all have the
OCI_ATTR_XID set. The global transaction ID should be the same.

errhp (IN)
The array of error handles. If OCI_SUCCESS is not returned, then these indicate which
branches received which errors.

Comments
Prepares the specified global transaction for commit. This call is valid only for
distributed transactions. This call is an advanced performance feature intended for use
only in situations where the caller is responsible for preparing all the branches in a
transaction.

Related Functions

OCITransPrepare()
OCITransPrepare()

Purpose
Prepares a 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 Functions
OCITransCommit(), OCITransForget()
OCITransRollback()

**Purpose**
Rolls back the current transaction.

**Syntax**
```c
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 Functions**
- OCITransCommit()
OCITransStart()

Purpose

Sets the beginning of a transaction.

Syntax

```c
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

Example 17–4 and Example 17–5 demonstrate the use of OCI transactional calls for manipulating global transactions. The concept for a single session operating on different branches, as shown in Example 17–4, is illustrated by Figure 8–2.

Example 17–4  Using OCITransStart() in a Single Session Operating on Different Branches

```c
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 **)&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);

    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);
}

Example 17–5 Using OCITransStart() in a Single Session Operating on Multiple Branches Sharing the Same Transaction

```c
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);
    OCIReturn(OCIEnvCreate(&envhp, OCI_DEFAULT, (void *)0, 0, 0, 0,
        (size_t)0, (void *)0);
    OCIReturn(OCIHandleAlloc( (void *) envhp, (void **) &errhp, (ub4) OCI_HTYPE_ERROR,
        52, (void **) &tmp);
    OCIReturn(OCIHandleAlloc( (void *) envhp, (void **) &srvhp, (ub4) OCI_HTYPE_SERVER,
        52, (void **) &tmp);
    OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
    OCIReturn(OCIHandleAlloc( (void *) envhp, (void **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
        52, (void **) &tmp);
    OCIReturn(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 Functions

OCITransDetach()}
Table 17–7 lists the miscellaneous OCI functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIBreak()&quot; on page 17-163</td>
<td>Perform an immediate asynchronous break</td>
</tr>
<tr>
<td>&quot;OCIClientVersion()&quot; on page 17-164</td>
<td>Return the client library version</td>
</tr>
<tr>
<td>&quot;OCIErrorGet()&quot; on page 17-165</td>
<td>Return error message and Oracle error</td>
</tr>
<tr>
<td>&quot;OCILdaToSvcCtx()&quot; on page 17-168</td>
<td>Toggle Lda_Def to service context handle</td>
</tr>
<tr>
<td>&quot;OCIPasswordChange()&quot; on page 17-169</td>
<td>Change password</td>
</tr>
<tr>
<td>&quot;OCIPing()&quot; on page 17-171</td>
<td>Confirm that the connection and the server are active</td>
</tr>
<tr>
<td>&quot;OCIReset()&quot; on page 17-172</td>
<td>Call after OCIBreak() to reset asynchronous operation and protocol</td>
</tr>
<tr>
<td>&quot;OCIRowidToChar()&quot; on page 17-173</td>
<td>Convert a Universal ROWID to character extended (base 64) representation</td>
</tr>
<tr>
<td>&quot;OCIUserCallbackGet()&quot; on page 17-177</td>
<td>Identify the callback that is registered for handle</td>
</tr>
<tr>
<td>&quot;OCIUserCallbackRegister()&quot; on page 17-179</td>
<td>Register a user-created callback function</td>
</tr>
<tr>
<td>&quot;OCIUserVersion()&quot; on page 17-175</td>
<td>Get the Oracle version string</td>
</tr>
<tr>
<td>&quot;OCISvcCtxToLda()&quot; on page 17-176</td>
<td>Toggle service context handle to Lda_Def</td>
</tr>
<tr>
<td>&quot;OCIUserVersion()&quot; on page 17-174</td>
<td>Get the Oracle release string</td>
</tr>
</tbody>
</table>
OCIBreak()

Purpose
Perform 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.

See Also:
- "Server Handle Attributes" on page A-11
- "Nonblocking Mode in OCI" on page 2-27
- "Canceling Calls" on page 2-25

Related Functions
OCIReset()
OCI Client Version

Purpose

Returns the 5 digit Oracle Database version number of the client library at run time.

Syntax

```c
void OCIClientVersion ( sword        *major_version,
                        sword        *minor_version,
                        sword        *update_num,
                        sword        *patch_num,
                        sword        *port_update_num );
```

Parameters

- **major_version (OUT)**
  The major version.

- **minor_version (OUT)**
  The minor version.

- **update_num (OUT)**
  The update number.

- **patch_num (OUT)**
  The patch number that was applied to the library.

- **port_update_num (OUT)**
  The port-specific patch applied to the library.

Comments

OCI Client Version returns the version 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.

In addition to OCIClientVersion() there are two macros defined: OCI_MAJOR_VERSION and OCI_MINOR_VERSION. These are useful for writing a generic application that can be built and run with different versions of OCI client. For example:

```c
....
#if (OCI_MAJOR_VERSION > 9)
...
#endif
....
```

Related Functions

- OCIServerRelease()
OCIDataErrorGet()

Purpose

Returns an error message in the buffer provided and an Oracle Database error code.

Syntax

```c
sword OCIDataErrorGet ( 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 OCIDataErrorGet() is OCI_ERROR. The client can then specify a bigger buffer and call OCIDataErrorGet() again.

  If bufsiz is sufficient to hold the entire message text and the message could be successfully copied into bufp, the return code for OCIDataErrorGet() is OCI_SUCCESS.

- **type (IN)**
  The type of the handle (OCI_HTYPE_ERROR or OCI_HTYPE_ENV).

Comments

This function does not support SQL statements. 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. This function can be called multiple times if there are multiple diagnostic records for an error.
Note that OCIErrorGet() must not be called when the return code is OCI_SUCCESS. Otherwise, an error message from a previously executed statement is found by OCIErrorGet().

The error handle is originally allocated with a call to OCIHandleAlloc().

---

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

---

**See Also:** "Error Handling in OCI" on page 2-20

---

**Example**

Example 17–6 shows a simplified example of a function for error checking using OCIErrorGet().

**Example 17–6 Using OCIErrorGet() for Error Checking**

```c
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 Functions

OCIHandleAlloc()
OCILdaToSvcCtx()

Purpose

Converts a V7 Lda_Def to a V8 or later service context handle.

Syntax

\[
\text{sword OCILdaToSvcCtx} \ ( \text{OCISvcCtx} \ *svchpp, \\
\text{OCIError} \ *\text{errhp}, \\
\text{Lda_def} \ *\text{ldap} );
\]

Parameters

\begin{itemize}
\item \textbf{svchpp (IN/OUT)}
The service context handle.
\item \textbf{errhp (IN/OUT)}
An error handle that you can pass to OCIErrorGet() for diagnostic information when there is an error.
\item \textbf{ldap (IN/OUT)}
The Oracle7 logon data area returned by OCISvcCtxToLda() from this service context.
\end{itemize}

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.

See Also: Appendix A, "Handle and Descriptor Attributes"

Related Functions

OCISvcCtxToLda()
OCIPasswordChange()

Purpose

Allows the password of an account to be changed.

Syntax

```c
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_PasswordChange()

- **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.

**Comments**

This call allows the password of an account to be changed. This call is similar to `OCI_SessionBegin()` 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 `OCI_SessionBegin()` returns an error (ORA-28001) or warning that indicates that the password has expired.

The `mode` or the environment handle determines if UTF-16 is being used.

**Related Functions**

`OCI_SessionBegin()`
OCIPing()

Purpose

Makes a round-trip call to the server to confirm that the connection and the server are active.

Syntax

sword OCIPing ( OCISvcCtx *svchp,
OCIError *errhp,
ub4 mode );

Parameters

svchp (IN)
A handle to a service context. The service context handle must be initialized and have a server context handle associated with it.

errhp (IN)
An error handle 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.

Related Functions

OCIHandleFree()
OCIReset()

Purpose

Resets the interrupted asynchronous operation and protocol. Must be called if an OCIBreak() call was issued while a nonblocking operation was in progress.

Syntax

```c
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 Functions

OCIBreak()
OCIRowidToChar()

Purpose

Converts a Universal ROWID to character extended (base 64) representation.

Syntax

```c
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.
OCIServerRelease() returns the Oracle Database release string.

Syntax

```c
sword OCIServerRelease ( void         *hndlp,
                         OCIError     *errhp,
                         OraText      *bufp,
                         ub4          bufsize
                         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:

```c
#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) & 0x000000FF))    /* port release number */
#define PORT_UPDATE_NUMPUP(v) ((sword)(((v) >> 0) & 0x000000FF)) /* port update number */
```

Related Functions

OCIServerVersion()
OCIServerVersion()

Purpose

Returns the Oracle Database version string.

Syntax

```c
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 Functions

**OCIErrorGet(), OCIClientVersion()**
OCISvcCtxToLda()

Purpose

Toggles between a V8 or later service context handle and a V7 Lda_Def.

Syntax

sword OCISvcCtxToLda ( OCISvcCtx    *srvhp,
                        OCIError     *errhp,
                        Lda_Def      *ldap );

Parameters

svchp (IN/OUT)
The service context handle.

errhp (IN/OUT)
An error handle 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.

See Also: Appendix A, "Handle and Descriptor Attributes"

Related Functions

OCILdaToSvcCtx()
OCIUserCallbackGet()

Purpose

Determines the callback that is registered for a handle.

Syntax

```c
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 17–8.

**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()" on page 17-179.
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()" on page 17-179 for information about the parameters of callbackp

cctxpp (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.

See Also: "Restrictions on Callback Functions" on page 9-35

Related Functions
OCIUserCallbackRegister()
**OC IUserCallbackRegister()**

**Purpose**

Registers a user-created callback function.

**Syntax**

```c
sword OC IUserCallbackRegister ( 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
OCIUserCallbackRegister()

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.

cctxp (IN)
A context pointer for the callback.

fcode (IN)
A unique function code of an OCI function. These are listed in Table 17–8.

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:

```c
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 17–8. Note that callbacks can be registered for only the OCI calls listed in Table 17–3.

- `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.
OCIUserCallbackRegister()

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: Appendix B, "OCI Demonstration Programs"

Table 17–8 and Table 17–9 list the OCI Function codes and provides the OCI routine name and its function number.

**Table 17–8 OCI Function Codes**

<table>
<thead>
<tr>
<th>#</th>
<th>OCI Routine</th>
<th>#</th>
<th>OCI Routine</th>
<th>#</th>
<th>OCI Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OCIInitialize</td>
<td>33</td>
<td>OCITransStart</td>
<td>65</td>
<td>OCIDefineByPos</td>
</tr>
<tr>
<td>2</td>
<td>OCIHandleAlloc</td>
<td>34</td>
<td>OCITransDetach</td>
<td>66</td>
<td>OCIBindByPos</td>
</tr>
<tr>
<td>3</td>
<td>OCIHandleFree</td>
<td>35</td>
<td>OCITransCommit</td>
<td>67</td>
<td>OCIBindByName</td>
</tr>
<tr>
<td>4</td>
<td>OCIDescriptorAlloc</td>
<td>36</td>
<td>(not used)</td>
<td>68</td>
<td>OCIlobAssign</td>
</tr>
<tr>
<td>5</td>
<td>OCIDescriptorFree</td>
<td>37</td>
<td>OCIErrorGet</td>
<td>69</td>
<td>OCIlobIsEqual</td>
</tr>
<tr>
<td>6</td>
<td>OCIEnvInit</td>
<td>38</td>
<td>OCIlobOpen</td>
<td>70</td>
<td>OCIlobLocatorIsInit</td>
</tr>
<tr>
<td>7</td>
<td>OCIServerAttach</td>
<td>39</td>
<td>OCIlobClose</td>
<td>71</td>
<td>OCIlobEnableBuffering</td>
</tr>
<tr>
<td>8</td>
<td>OCIServerDetach</td>
<td>40</td>
<td>(not used)</td>
<td>72</td>
<td>OCIlobCharSetId</td>
</tr>
<tr>
<td>9</td>
<td>(not used)</td>
<td>41</td>
<td>(not used)</td>
<td>73</td>
<td>OCIlobCharSetForm</td>
</tr>
<tr>
<td>10</td>
<td>OCISessionBegin</td>
<td>42</td>
<td>OCIlobCopy</td>
<td>74</td>
<td>OCIlobFileSetName</td>
</tr>
<tr>
<td>11</td>
<td>OCISessionEnd</td>
<td>43</td>
<td>OCIlobAppend</td>
<td>75</td>
<td>OCIlobFileGetName</td>
</tr>
<tr>
<td>12</td>
<td>OCIPasswordChange</td>
<td>44</td>
<td>OCIlobErase</td>
<td>76</td>
<td>OCIlogon</td>
</tr>
<tr>
<td>13</td>
<td>OCISmtPrepare</td>
<td>45</td>
<td>OCIlobGetLength</td>
<td>77</td>
<td>OCIlogoff</td>
</tr>
<tr>
<td>14</td>
<td>(not used)</td>
<td>46</td>
<td>OCIlobTrim</td>
<td>78</td>
<td>OCIlobDisableBuffering</td>
</tr>
<tr>
<td>15</td>
<td>(not used)</td>
<td>47</td>
<td>OCIlobRead</td>
<td>79</td>
<td>OCIlobFlushBuffer</td>
</tr>
<tr>
<td>16</td>
<td>(not used)</td>
<td>48</td>
<td>OCIlobWrite</td>
<td>80</td>
<td>OCIlobLoadFromFile</td>
</tr>
<tr>
<td>17</td>
<td>OCIBindDynamic</td>
<td>49</td>
<td>(not used)</td>
<td>81</td>
<td>OCIlobOpen</td>
</tr>
<tr>
<td>18</td>
<td>OCIBindObject</td>
<td>50</td>
<td>OCIBreak</td>
<td>82</td>
<td>OCIlobClose</td>
</tr>
<tr>
<td>19</td>
<td>(not used)</td>
<td>51</td>
<td>OCIServerVersion</td>
<td>83</td>
<td>OCIlobIsOpen</td>
</tr>
<tr>
<td>20</td>
<td>OCIBindArrayOfStruct</td>
<td>52</td>
<td>(not used)</td>
<td>84</td>
<td>OCIlobFilesOpen</td>
</tr>
<tr>
<td>21</td>
<td>OCISmtExecute</td>
<td>53</td>
<td>(not used)</td>
<td>85</td>
<td>OCIlobFileExists</td>
</tr>
<tr>
<td>22</td>
<td>(not used)</td>
<td>54</td>
<td>OCIAttrGet</td>
<td>86</td>
<td>OCIlobFileCloseAll</td>
</tr>
<tr>
<td>23</td>
<td>(not used)</td>
<td>55</td>
<td>OCIAttrSet</td>
<td>87</td>
<td>OCIlobCreateTemporary</td>
</tr>
<tr>
<td>24</td>
<td>(not used)</td>
<td>56</td>
<td>OCIParamSet</td>
<td>88</td>
<td>OCIlobFreeTemporary</td>
</tr>
<tr>
<td>25</td>
<td>OCIDefineObject</td>
<td>57</td>
<td>OCIParamGet</td>
<td>89</td>
<td>OCIlobIsTemporary</td>
</tr>
</tbody>
</table>
Table 17–8  (Cont.) OCI Function Codes

<table>
<thead>
<tr>
<th>#</th>
<th>OCI Routine</th>
<th>#</th>
<th>OCI Routine</th>
<th>#</th>
<th>OCI Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>OCIDefineDynamic</td>
<td>58</td>
<td>OCIStmtGetTypeInfo</td>
<td>90</td>
<td>OCIAQEnq</td>
</tr>
<tr>
<td>27</td>
<td>OCIDefineArrayOfStruct</td>
<td>59</td>
<td>OCIStmtSetTypeInfo</td>
<td>91</td>
<td>OCIAQDeq</td>
</tr>
<tr>
<td>28</td>
<td>OCIStmtFetch</td>
<td>60</td>
<td>(not used)</td>
<td>92</td>
<td>OCIRest</td>
</tr>
<tr>
<td>29</td>
<td>OCIStmtGetBindInfo</td>
<td>61</td>
<td>OCIStmtSetTypeInfo</td>
<td>93</td>
<td>OCISvcCtxToLda</td>
</tr>
<tr>
<td>30</td>
<td>(not used)</td>
<td>62</td>
<td>OCITransForget</td>
<td>94</td>
<td>OCILobLocatorAssign</td>
</tr>
<tr>
<td>31</td>
<td>(not used)</td>
<td>63</td>
<td>OCITransPrepare</td>
<td>95</td>
<td>(not used)</td>
</tr>
<tr>
<td>32</td>
<td>OCIDescribeAny</td>
<td>64</td>
<td>OCITransRollback</td>
<td>96</td>
<td>OCIAQListen</td>
</tr>
</tbody>
</table>

Table 17–9  Continuation of OCI Function Codes from 97 and Higher

<table>
<thead>
<tr>
<th>#</th>
<th>OCI Routine</th>
<th>#</th>
<th>OCI Routine</th>
<th>#</th>
<th>OCI Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
<td>Reserved</td>
<td>113</td>
<td>OCILobErase2</td>
<td>129</td>
<td>OCILobGetOptions</td>
</tr>
<tr>
<td>98</td>
<td>Reserved</td>
<td>114</td>
<td>OCILobGetLength2</td>
<td>130</td>
<td>OCILobSetOptions</td>
</tr>
<tr>
<td>99</td>
<td>OCITransMultiPrepare</td>
<td>115</td>
<td>OCILobLoadFromFile2</td>
<td>131</td>
<td>OCILobFragmentInsert</td>
</tr>
<tr>
<td>100</td>
<td>OCIConnectionPoolCreate</td>
<td>116</td>
<td>OCILobRead2</td>
<td>132</td>
<td>OCILobFragmentDelete</td>
</tr>
<tr>
<td>101</td>
<td>OCIConnectionPoolDestroy</td>
<td>117</td>
<td>OCILobTrim2</td>
<td>133</td>
<td>OCILobFragmentMove</td>
</tr>
<tr>
<td>102</td>
<td>OCILogon2</td>
<td>118</td>
<td>OCILobWrite2</td>
<td>134</td>
<td>OCILobFragmentReplace</td>
</tr>
<tr>
<td>103</td>
<td>OCIRowidToChar</td>
<td>119</td>
<td>OCILobGetStorageLimit</td>
<td>135</td>
<td>OCILobGetDeduplicateRegions</td>
</tr>
<tr>
<td>104</td>
<td>OCISessionPoolCreate</td>
<td>120</td>
<td>OCIDBStartup</td>
<td>136</td>
<td>OCIAppCtxSet</td>
</tr>
<tr>
<td>105</td>
<td>OCISessionPoolDestroy</td>
<td>121</td>
<td>OCIDBShutdown</td>
<td>137</td>
<td>OCIAppCtxClearAll</td>
</tr>
<tr>
<td>106</td>
<td>OCISessionGet</td>
<td>122</td>
<td>OCILobArrayRead</td>
<td>138</td>
<td>OCILobGetContentType</td>
</tr>
<tr>
<td>107</td>
<td>OCISessionRelease</td>
<td>123</td>
<td>OCILobArrayWrite</td>
<td>139</td>
<td>OCILobSetContentType</td>
</tr>
<tr>
<td>108</td>
<td>OCIStmtPrepare2</td>
<td>124</td>
<td>OCIAQEnqStreaming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>OCIStmtRelease</td>
<td>125</td>
<td>OCIAQGetReplayInfo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>OCIAQEnqArray</td>
<td>126</td>
<td>OCIAQResetReplayInfo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>OCIAQDeqArray</td>
<td>127</td>
<td>OCIArrayDescriptorAlloc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>OCILobCopy2</td>
<td>128</td>
<td>OCIArrayDescriptorFree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Related Functions

OCIUserCallbackGet()
OCIUserCallbackRegister()
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 Appendix B.

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

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: Chapter 11 and Chapter 14 for more information about using the calls in this chapter

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.
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 18–1 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).

The distinction between various instances is further described in Table 18–1.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Persistent Object</th>
<th>Transient Object</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>object type</td>
<td>object type</td>
<td>object type, built-in, collection</td>
</tr>
<tr>
<td>Maximum Lifetime</td>
<td>until object is deleted</td>
<td>session</td>
<td>session</td>
</tr>
<tr>
<td>Referenceable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Embeddable</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
**Terminology**

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 an 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" on page 11-3 for further discussion of the terms used to refer to different types of objects

**Conventions for OCI Functions**

See the "Conventions for OCI Functions" on page 16-1 for the conventions used in describing each function. The entries for each function may also contain the following information:

**Return Values**

A description of what value is returned by the function if the function returns something other than the standard codes listed in Table 18–3.

**Navigational Function Return Values**

Table 18–2 lists the values that OCI navigational functions typically return.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_SUCCESS</td>
<td>The operation succeeded.</td>
</tr>
<tr>
<td>OCI_ERROR</td>
<td>The operation failed. The specific error can be retrieved by calling OCIErrorGet() on the error handle passed to the function.</td>
</tr>
<tr>
<td>OCI_INVALID_HANDLE</td>
<td>The OCI handle passed to the function is invalid.</td>
</tr>
</tbody>
</table>

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" on page 2-20 for more information about return codes and error handling

**Server Round-Trips for Cache and Object Functions**

For a table showing the number of server round-trips required for individual OCI cache and object functions, see Table C–3.
Navigational Function Error Codes

Table 18–3 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>
<thead>
<tr>
<th>Function</th>
<th>Possible ORA Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCICacheFlush()</td>
<td>24350, 21560, 21705</td>
</tr>
<tr>
<td>OCICacheFree()</td>
<td>24350, 21560, 21705</td>
</tr>
<tr>
<td>OCICacheRefresh()</td>
<td>24350, 21560, 21705</td>
</tr>
<tr>
<td>OCICacheUnmark()</td>
<td>24350, 21560, 21705</td>
</tr>
<tr>
<td>OCICacheUnpin()</td>
<td>24350, 21560, 21705</td>
</tr>
<tr>
<td>OCIObjectArrayPin()</td>
<td>24350, 21560</td>
</tr>
<tr>
<td>OCIObjectCopy()</td>
<td>24350, 21560, 21705, 21710</td>
</tr>
<tr>
<td>OCIObjectExists()</td>
<td>24350, 21560, 21710</td>
</tr>
<tr>
<td>OCIObjectFlush()</td>
<td>24350, 21560, 21701, 21703, 21708, 21710</td>
</tr>
<tr>
<td>OCIObjectFree()</td>
<td>24350, 21560, 21603, 21710</td>
</tr>
<tr>
<td>OCIObjectGetAttr()</td>
<td>21560, 21600, 22305</td>
</tr>
<tr>
<td>OCIObjectGetInd()</td>
<td>24350, 21560, 21710</td>
</tr>
<tr>
<td>OCIObjectGetObjectRef()</td>
<td>24350, 21560, 21710</td>
</tr>
<tr>
<td>OCIObjectGetTypeRef()</td>
<td>24350, 21560, 21710</td>
</tr>
<tr>
<td>OCIObjectIsDirty()</td>
<td>24350, 21560, 21710</td>
</tr>
<tr>
<td>OCIObjectIsLocked()</td>
<td>24350, 21560, 21710</td>
</tr>
<tr>
<td>OCIObjectLock()</td>
<td>24350, 21560, 21701, 21708, 21710</td>
</tr>
<tr>
<td>OCIObjectLockNoWait()</td>
<td>24350, 21560, 21701, 21708, 21710</td>
</tr>
<tr>
<td>OCIObjectMarkDelete()</td>
<td>24350, 21560, 21700, 21701, 21702, 21710</td>
</tr>
<tr>
<td>OCIObjectMarkDeleteByRef()</td>
<td>24350, 21560</td>
</tr>
<tr>
<td>OCIObjectMarkUpdate()</td>
<td>24350, 21560, 21700, 21701, 21710</td>
</tr>
<tr>
<td>OCIObjectNew()</td>
<td>24350, 21560, 21705, 21710</td>
</tr>
<tr>
<td>OCIObjectPin()</td>
<td>24350, 21560, 21700, 21702</td>
</tr>
<tr>
<td>OCIObjectPinCountReset()</td>
<td>24350, 21560, 21710</td>
</tr>
<tr>
<td>OCIObjectPinTable()</td>
<td>24350, 21560, 21705</td>
</tr>
<tr>
<td>OCIObjectRefresh()</td>
<td>24350, 21560, 21709, 21710</td>
</tr>
<tr>
<td>OCIObjectSetAttr()</td>
<td>21560, 21600, 22305, 22279, 21601</td>
</tr>
<tr>
<td>OCIObjectUnmark()</td>
<td>24350, 21560, 21710</td>
</tr>
<tr>
<td>OCIObjectUnmarkByRef()</td>
<td>24350, 21560</td>
</tr>
<tr>
<td>OCIObjectUnpin()</td>
<td>24350, 21560, 21710</td>
</tr>
</tbody>
</table>

The ORA errors in Table 18–3 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
OCI Flush or Refresh Functions

Table 18–4 describes the OCI flush or refresh functions that are described in this section.

Table 18–4  Flush or Refresh Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCICacheFlush()&quot; on page 18-7</td>
<td>Flush modified persistent objects in cache to server</td>
</tr>
<tr>
<td>&quot;OCICacheRefresh()&quot; on page 18-9</td>
<td>Refresh pinned persistent objects</td>
</tr>
<tr>
<td>&quot;OCIObjectFlush()&quot; on page 18-11</td>
<td>Flush a modified persistent object to the server</td>
</tr>
<tr>
<td>&quot;OCIObjectRefresh()&quot; on page 18-12</td>
<td>Refresh a persistent object</td>
</tr>
</tbody>
</table>
**OCIFlush or Refresh Functions**

**OCI Navigational and Type Functions**

18-7

---

**OCICacheFlush()**

**Purpose**
Flushes modified persistent objects to the server.

**Syntax**

```c
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 newed or marked as updated or marked as deleted.

See Also: "OCIObjectFlush()" on page 18-11

This function incurs, at most, one network round-trip.

Related Functions

OCIObjectFlush()
**OCIRefresh**

**Purpose**

Refreshes all pinned persistent objects in the cache.

**Syntax**

```c
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.

See Also: 
- OCIObjectRefresh()
- "Refreshing an Object Copy" on page 14-9.

Caution: When objects are refreshed, the secondary-level memory of those objects could potentially move to a different place in memory. As a result, any pointers to attributes that were saved prior to this call may be invalidated. Examples of attributes using secondary-level memory include OCIString *, OCIColl *, and OCIRaw *.

Related Functions

OCIObjectRefresh()
**OCIObjectFlush()**

**Purpose**
Flushes a modified persistent object to the server.

**Syntax**
```
sword OCIObjectFlush ( OCIEnv *env, 
                      OCIError *err, 
                      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.

**See Also:** "LOB Functions" on page 17-17

**Related Functions**
- `OCIObjectPin()`, `OCICacheFlush()`
OCIObjectRefresh()

Purpose

Refreshes a persistent object from the most current database snapshot.

Syntax

```c
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.

---

**Caution:** Modifications made to objects (dirty objects) since the last flush are lost if unmarked objects are refreshed by this function.

---

Table 18–5 shows how the various meta-attribute flags and durations of an object are modified after being refreshed.

**Table 18–5  Object Status After Refresh**

<table>
<thead>
<tr>
<th>Object Attribute</th>
<th>Status After Refresh</th>
</tr>
</thead>
<tbody>
<tr>
<td>existent</td>
<td>Set to appropriate value</td>
</tr>
<tr>
<td>pinned</td>
<td>Unchanged</td>
</tr>
<tr>
<td>allocation duration</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>
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 Functions

- OCICacheRefresh()
Table 18–6 describes the OCI mark or unmark object and cache functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCICacheUnmark()&quot; on page 18-15</td>
<td>Unmark objects in the cache</td>
</tr>
<tr>
<td>&quot;OCIObjectMarkDelete()&quot; on page 18-16</td>
<td>Mark an object deleted or delete a value instance</td>
</tr>
<tr>
<td>&quot;OCIObjectMarkDeleteByRef()&quot; on page 18-17</td>
<td>Mark an object deleted when given a reference to it</td>
</tr>
<tr>
<td>&quot;OCIObjectMarkUpdate()&quot; on page 18-18</td>
<td>Mark an object as updated or dirty</td>
</tr>
<tr>
<td>&quot;OCIObjectUnmark()&quot; on page 18-19</td>
<td>Unmark an object</td>
</tr>
<tr>
<td>&quot;OCIObjectUnmarkByRef()&quot; on page 18-20</td>
<td>Unmark an object, when given a reference to it</td>
</tr>
</tbody>
</table>
OCI Cache Unmark

**Purpose**
Unmarks all dirty objects in the object cache.

**Syntax**
```c
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.

**See Also:** "OCIObjectUnmark()" on page 18-19 for more information about unmarking an object

**Related Functions**
- OCIOBJECTUnmark()
OCIObjectMarkDelete()  

Purpose  
Marks a standalone instance as deleted, when given a pointer to the instance.

Syntax  

```c
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 Functions  
OCIObjectMarkDeleteByRef(), OCIObjectGetProperty()
**OCIObjectMarkDeleteByRef()**

**Purpose**
Marks an object as deleted, when given a reference to the object.

**Syntax**
```c
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 Functions**
- `OCIObjectMarkDelete()`, `OCIObjectGetProperty()`
OCIObjectMarkUpdate()

Purpose
Marks a persistent object as updated (dirty).

Syntax

```c
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.

See Also:  "Marking Objects and Flushing Changes" on page 11-10 for more information about the use of this function

Related Functions

OCIObjectPin(), OCIObjectGetProperty(), OCIObjectIsDirty(), OCIObjectUnmark()
OCIObjectUnmark()

Purpose

Unmarks an object as dirty.

Syntax

```c
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 Functions

- `OCIObjectUnmarkByRef()`
OCIObjectUnmarkByRef()

Purpose

Unmarks an object as dirty, when given a ref to the object.

Syntax

```c
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 Functions

OCIObjectUnmark()
OCI Get Object Status Functions

Table 18–7 describes the OCI get object status functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIObjectExists()&quot; on page 18-22</td>
<td>Get the existent status of an instance</td>
</tr>
<tr>
<td>&quot;OCIObjectGetProperty()&quot; on page 18-23</td>
<td>Get the status of a particular object property</td>
</tr>
<tr>
<td>&quot;OCIObjectIsDirty()&quot; on page 18-26</td>
<td>Get the dirtied status of an instance</td>
</tr>
<tr>
<td>&quot;OCIObjectIsLocked()&quot; on page 18-27</td>
<td>Get the locked status of an instance</td>
</tr>
</tbody>
</table>
OCIObjectExists()

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.

See Also: "Object Meta-Attributes" on page 11-12

Related Functions

OCIObjectPin()
**OCIObjectGetProperty()**

**Purpose**
Retrieves a given property of an object.

**Syntax**
```c
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" on page 14-11.

**OCI_OBJECTPROP_ALLOC_DURATION**
This returns the allocation duration of the object. The property argument must be a pointer to a variable of type OCIDuration. Valid values include:
- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

For more information about durations, see "Object Duration" on page 14-11.

**OCI_OBJECTPROP_LOCK**
This returns the lock status of the object. The possible lock statuses are enumerated by OCILockOpt. An error is returned if the given object points to a transient or value instance. The property argument must be a pointer to a variable of type OCILockOpt. 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 Get Object Status Functions

- **OCI_LOCK_X_NOWAIT** (exclusive lock with the NOWAIT option)

  **See Also:** "Locking with the NOWAIT Option" on page 14-10

**OCI_OBJECTPROP_MARKSTATUS**
This returns the dirty status and indicates whether the object is a new object, updated object, or deleted object. An error is returned if the given object points to a transient or value instance. The property argument must be of type `OCIObjectMarkStatus`. Valid values include:

- `OCI_OBJECT_NEW`
- `OCI_OBJECT_DELETED`
- `OCI_OBJECT_UPDATED`

The following macros are available to test the object mark status:

- `OCI_OBJECT_IS_UPDATED` (flag)
- `OCI_OBJECT_IS_DELETED` (flag)
- `OCI_OBJECT_IS_NEW` (flag)
- `OCI_OBJECT_IS_DIRTY` (flag)

**OCI_OBJECTPROP_VIEW**
This identifies whether the specified object is a view object or not. If the property value returned is `TRUE`, the object is a view; otherwise, it is not. An error is returned if the given object points to a transient or value instance. The property argument must be of type `boolean`.

**Related Functions**

- `OCIObjectLock()`, `OCIObjectMarkDelete()`, `OCIObjectMarkUpdate()`, `OCIObjectPin()`, `OCIObjectPin()`
OCIObjectIsDirty()

Purpose

Checks to see if an object is marked as dirty.

Syntax

```c
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 Functions

- `OCIObjectMarkUpdate()`, `OCIObjectGetProperty()`
OCIObjectIsLocked()

Purpose

Gets lock status of an object.

Syntax

```c
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 Functions

OCIObjectLock(), OCIObjectGetProperty()
OCI Miscellaneous Object Functions

Table 18–8 describes the miscellaneous object functions that are described in this section.

Table 18–8  Miscellaneous Object Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIObjectCopy()&quot; on page 18-29</td>
<td>Copy one instance to another</td>
</tr>
<tr>
<td>&quot;OCIObjectGetAttr()&quot; on page 18-31</td>
<td>Get an object attribute</td>
</tr>
<tr>
<td>&quot;OCIObjectGetInd()&quot; on page 18-33</td>
<td>Get NULL structure of an instance</td>
</tr>
<tr>
<td>&quot;OCIObjectGetObjectRef()&quot; on page 18-34</td>
<td>Return reference to a given object</td>
</tr>
<tr>
<td>&quot;OCIObjectGetTypeRef()&quot; on page 18-35</td>
<td>Get a reference to a TDO of an instance</td>
</tr>
<tr>
<td>&quot;OCIObjectLock()&quot; on page 18-36</td>
<td>Lock a persistent object</td>
</tr>
<tr>
<td>&quot;OCIObjectLockNoWait()&quot; on page 18-37</td>
<td>Lock a persistent object but do not wait for the lock</td>
</tr>
<tr>
<td>&quot;OCIObjectNew()&quot; on page 18-38</td>
<td>Create a new instance</td>
</tr>
<tr>
<td>&quot;OCIObjectSetAttr()&quot; on page 18-41</td>
<td>Set an object attribute</td>
</tr>
</tbody>
</table>
OCIObjectCopy()

Purpose

Copies a source instance to a destination.

Syntax

```c
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()" on page 18-50

**null_source (IN)**
Pointer to the NULL structure of the source object.

**target (IN)**
A pointer to the target instance; if it is an object, 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:

- If the option OCI_OBJECTCOPY_NOREF is specified in the option parameter, then all references in the source are not copied. Instead, the references in the target are set to NULL.
- If the attribute is an internal LOB, then only the LOB locator from the source object is copied. A copy of the LOB data is not made until OCIObjectFlush() is called. Before the target object is flushed, both the source and the target locators refer to the same LOB value.

The target or the containing instance of the target must 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 Functions
OCIObjectPin()
OCIObjectGetAttr()

Purpose

Retrieves an object attribute.

Syntax

```c
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 (IN)**
  Pointer to a `OCIInd` to indicate if an attribute is null.

- **attr_null_struct (IN)**
  Pointer to a `void *` to return the NULL structure.

- **attr_value (IN)**
  Pointer to a `void *` to return the attribute value.

- **attr_tdo (IN)**
  Pointer to a `struct OCIType *` to return the type descriptor object (TDO).
**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.

**attr_tdo** *(OUT)*

Pointer to the TDO of the attribute.

**Comments**

This function gets a value from an object or from an array. If the parameter `instance` points to an object, then the path expression specifies the location of the attribute in the object. It is assumed that the object is pinned and that the value returned is valid until the object is unpinned.

If both `attr_null_status` and `attr_null_struct` are NULL, no NULL information is returned.

**Related Functions**

OCIObjectSetAttr()
OCIObjectGetInd()

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" on page 11-21 for a
discussion of the NULL indicator structure and examples of its use

Comments

None.

Related Functions

OCIObjectPin()
OCIObjectGetObjectRef()

Purpose

Returns a reference to a given persistent object.

Syntax

```c
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.

See Also:  "Object Meta-Attributes" on page 11-12

Related Functions

OCIObjectPin()
OCIObjectGetTypeRef()

Purpose

Returns a reference to the type descriptor object (TDO) of a standalone instance.

Syntax

```
sword OCIObjectGetTypeRef ( OCIEnv *env,
                           OCIError *err,
                           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 Functions

- OCIObjectPin()
OCIObjectLock()

Purpose

Locks a persistent object at the server.

Syntax

```c
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.

See Also:  "Locking Objects for Update" on page 14-10

Related Functions

OCIObjectPin(), OCIObjectIsLocked(), OCIObjectGetProperty(),
OCIObjectLockNoWait()
OCIObjectLockNoWait()

Purpose

Locks a persistent object at the server but does not wait for the lock. Returns an error if the lock is unavailable.

Syntax

```c
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.

See Also: "Locking Objects for Update" on page 14-10

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 Functions

OCIObjectPin(), OCIObjectIsLocked(), OCIObjectGetProperty(), OCIObjectLock()
OCIOBJECTNEW()

Purpose

Creates a standalone instance.

Syntax

```c
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" on page 3-25

**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 18–9 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. It can create an OCIString object with a Unicode buffer if the typecode indicates the object to be created is OCIString.

See Also: "Typecodes" on page 3-25

Table 18–9 shows that based on the parameters typecode (or tdo), value, and table, different instances are created.

<table>
<thead>
<tr>
<th>Type of the Instance</th>
<th>Table != NULL</th>
<th>Table == NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>object type (value=TRUE)</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>object type (value=FALSE)</td>
<td>persistent object</td>
<td>transient object</td>
</tr>
<tr>
<td>built-in type</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>collection type</td>
<td>value</td>
<td>value</td>
</tr>
</tbody>
</table>

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" on page 11-25 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" on page 11-24

Objects with LOB Attributes

If the object contains an internal LOB attribute, the LOB is set to empty. The object must be marked as dirty and flushed (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 Functions

OCIObjectPinTable(), OCIObjectFree()
**OCIObjectSetAttr()**

**Purpose**
Sets an object attribute.

**Syntax**

```c
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`.  

---

OCI Navigational and Type Functions 18-41
**attr_null_status (IN)**
The NULL status of the attribute if the type of attribute is primitive.

**attr_null_struct (IN)**
The NULL structure of an object or collection attribute.

**attr_value (IN)**
Pointer to the attribute value.

**Comments**
This function sets the attribute of the given object with the given value. The position of the attribute is specified as a path expression, which is an array of names and an array of indexes.

**Example**
For the path expression stanford.cs.stu[5].addr, the arrays appear as:

- names = {"stanford", "cs", "stu", "addr"}
- lengths = {8, 2, 3, 4}
- indexes = {5}

**Related Functions**
OCIObjectGetAttr()
Table 18–10 describes the OCI pin, unpin, and free functions that are described in this section.

**Table 18–10 Pin, Unpin, and Free Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCICacheFree()&quot; on page 18-44</td>
<td>Free objects in the cache</td>
</tr>
<tr>
<td>&quot;OCICacheUnpin()&quot; on page 18-45</td>
<td>Unpin persistent objects in cache or connection</td>
</tr>
<tr>
<td>&quot;OCIObjectArrayPin()&quot; on page 18-46</td>
<td>Pin an array of references</td>
</tr>
<tr>
<td>&quot;OCIObjectFree()&quot; on page 18-48</td>
<td>Free a previously allocated object</td>
</tr>
<tr>
<td>&quot;OCIObjectPin()&quot; on page 18-50</td>
<td>Pin an object</td>
</tr>
<tr>
<td>&quot;OCIObjectPinCountReset()&quot; on page 18-52</td>
<td>Unpin an object to zero pin count</td>
</tr>
<tr>
<td>&quot;OCIObjectPinTable()&quot; on page 18-53</td>
<td>Pin a table object with a given duration</td>
</tr>
<tr>
<td>&quot;OCIObjectUnpin()&quot; on page 18-55</td>
<td>Unpin an object</td>
</tr>
</tbody>
</table>
OCICacheFree()

Purpose

Frees all objects and values in the cache for the specified connection.

Syntax

```c
sword OCICacheFree ( OCIEnv              *env,
          OCIErr               *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 `OCIErrGet()`.

- **svc (IN)**
  An OCI service context.

Comments

If a connection is specified, this function frees the persistent objects, transient objects and values allocated for that connection. Otherwise, all persistent objects, transient objects and values in the object cache are freed. Objects are freed regardless of their pin count.

See Also: "OCIObjectFree()" on page 18-48 for more information about freeing an instance

Related Functions

OCIObjectFree()
OCICacheUnpin()

Purpose

Unpins persistent objects.

Syntax

sword OCICacheUnpin ( OCIEnv              *env,
                     OCIError            *err,
                     const OCISvcCtx     *svc );

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the 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 OCIErrrorGet().

svc (IN)
An OCI service context handle. The objects on the specified connection are unpinned.

Comments

This function completely unpins all of the persistent objects for the given connection.
The pin count for the objects is reset to zero.

See Also:
- "Pinning an Object" on page 11-8
- "Pin Count and Unpinning" on page 11-21

Related Functions

OCIOBJECTUnpin()
OCIObjectArrayPin()

Purpose

Pins an array of references.

Syntax

```c
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()" on page 18-50

- **pin_duration (IN)**
Pin duration. See `OCIObjectPin()`.

- **lock (IN)**
Lock option. See `OCIObjectPin()`.
**obj_array (OUT)**
If this argument is not `NULL`, the pinned objects 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 Functions**
OCIObjectPin()
OCIObjectFree()

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_NONNULL 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.

Freening 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 Functions

OCICacheFree()
OCIObjectPin()

Purpose

Pins a referenceable object.

Syntax

```c
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: "Locking with the NOWAIT Option" on page 14-10
**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.

**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 Functions**
`OCIObjectUnpin()`, `OCIObjectPinCountReset()`
OCIObjectPinCountReset()

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.

See Also: "Pin Count and Unpinning" on page 11-21

Related Functions

OCIObjectPin(), OCIObjectUnpin()
**OCIObjectPinTable()**

**Purpose**

Pins a table object for a specified duration.

**Syntax**

```c
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()" on page 18-50

- **object (OUT)**
  The pinned table object.
Comments

This function pins a table object with the specified pin duration. The client can unpin the object by calling `OCIObjectUnpin()`.

The table object pinned by this call can be passed as a parameter to `OCIObjectNew()` to create a standalone persistent object.

---

**Note:** The TDO (array of TDOs or table definition) obtained by this function 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 Functions

`OCIObjectPin()`, `OCIObjectUnpin()`
OCIObjectUnpin()

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().
OCIObjectUnpin()

**For Values**
This function returns an error for values.

**Related Functions**
OCIObjectPin(), OCIObjectPinCountReset()
OCI Type Information Accessor Functions

Table 18–11 describes the OCI type information accessor functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCITypeArrayByName()&quot; on page 18-58</td>
<td>Get an array of TDOs when given an array of object names</td>
</tr>
<tr>
<td>&quot;OCITypeArrayByRef()&quot; on page 18-60</td>
<td>Get an array of TDOs when given an array of object references</td>
</tr>
<tr>
<td>&quot;OCITypeByName()&quot; on page 18-62</td>
<td>Get a TDO when given an object name</td>
</tr>
<tr>
<td>&quot;OCITypeByRef()&quot; on page 18-64</td>
<td>Get a TDO when given an object reference</td>
</tr>
</tbody>
</table>
OCITypeArrayByName()

Purpose

Gets an array of types when given an array of names.

Syntax

```c
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 Functions
OCITypeArrayByRef(), OCITypeByName(), OCITypeByRef()
OCITypeArrayByRef()

Purpose

Gets an array of types when given an array of references.

Syntax

```c
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 Functions**

`OCITypeArrayByName()`, `OCITypeByRef()`, `OCITypeByName()`
OCITypeByName()

Purpose

Gets the most current version of an existing type by name.

Syntax

```c
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
attempts 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" on page 14-11

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)

dto (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 Functions

OCITypeByRef(), OCITypeArrayByName(), OCITypeArrayByRef()
OCITypeByRef()

Purpose

Gets a type when given a reference.

Syntax

```c
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 OCILogEnvCreate(), OCILogEnvNlsCreate(), and OCILogInitialize() (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 OCILogErrorGet().

- **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 Functions

OCITypeByName(), OCITypeArrayByName(), OCITypeArrayByRef()
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 Appendix B.

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**

### Introduction to Data Type Mapping and Manipulation Functions

This chapter describes the OCI data type mapping and manipulation functions in detail.

**See Also:** Chapter 12 for more information about the functions listed in this chapter

### Conventions for OCI Functions

See the "Conventions for OCI Functions" on page 16-1 for the conventions used in describing each function. The entries for each function may also contain the following information:
Returns

A description of what value is returned by the function if the function returns something other than the standard return codes listed in "Function Return Values" on page 19-2.

Data Type Mapping and Manipulation Function Return Values

The OCI data type mapping and manipulation functions typically return one of the values shown in Table 19–1.

Table 19–1 Function Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_SUCCESS</td>
<td>The operation succeeded.</td>
</tr>
<tr>
<td>OCI_ERROR</td>
<td>The operation failed. The specific error can be retrieved by calling OCIErrorGet() on the error handle passed to the function.</td>
</tr>
<tr>
<td>OCI_INVALID_HANDLE</td>
<td>The OCI handle passed to the function is invalid.</td>
</tr>
</tbody>
</table>

See Also: "Error Handling in OCI" on page 2-20 for more information about return codes and error handling

Functions Returning Other Values

Some functions return values other than those listed in Table 19–1. 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()
- OCIRawHexSize()
- OCIRefIsEqual()
- OCIRefIsNull()
- OCIStringPtr()
- OCIStringSize()

Server Round-Trips for 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.

Examples

For more information about these functions, including some code examples, see Chapter 12.
OCI Collection and Iterator Functions

Table 19–2 describes the collection and iterator functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCICollAppend()&quot; on page 19-4</td>
<td>Append an element to the end of a collection</td>
</tr>
<tr>
<td>&quot;OCICollAssign()&quot; on page 19-5</td>
<td>Assign (deep copy) one collection to another</td>
</tr>
<tr>
<td>&quot;OCICollAssignElem()&quot; on page 19-6</td>
<td>Assign the given element value ( elem ) to the element at ( coll[index] )</td>
</tr>
<tr>
<td>&quot;OCICollGetElem()&quot; on page 19-7</td>
<td>Get pointer to an element</td>
</tr>
<tr>
<td>&quot;OCICollGetElemArray()&quot; on page 19-9</td>
<td>Get an array of elements from a collection</td>
</tr>
<tr>
<td>&quot;OCICollIsLocator()&quot; on page 19-11</td>
<td>Indicate whether a collection is locator-based or not</td>
</tr>
<tr>
<td>&quot;OCICollMax()&quot; on page 19-12</td>
<td>Return maximum number of elements in collection</td>
</tr>
<tr>
<td>&quot;OCICollSize()&quot; on page 19-13</td>
<td>Get current size of collection (in number of elements)</td>
</tr>
<tr>
<td>&quot;OCICollTrim()&quot; on page 19-15</td>
<td>Trim elements from the collection</td>
</tr>
<tr>
<td>&quot;OCIIterCreate()&quot; on page 19-16</td>
<td>Create iterator to scan the varray elements</td>
</tr>
<tr>
<td>&quot;OCIIterDelete()&quot; on page 19-17</td>
<td>Delete iterator</td>
</tr>
<tr>
<td>&quot;OCIIterGetCurrent()&quot; on page 19-18</td>
<td>Get current collection element</td>
</tr>
<tr>
<td>&quot;OCIIterInit()&quot; on page 19-19</td>
<td>Initialize iterator to scan the given collection</td>
</tr>
<tr>
<td>&quot;OCIIterNext()&quot; on page 19-20</td>
<td>Get next collection element</td>
</tr>
<tr>
<td>&quot;OCIIterPrev()&quot; on page 19-22</td>
<td>Get previous collection element</td>
</tr>
</tbody>
</table>
OCICollAppend()

Purpose

Appends an element to the end of a collection.

Syntax

```c
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 Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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.

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 Functions

- OCIErrorGet()
OCICollAssign()

Purpose
Assigns (deep copies) one collection to another.

Syntax
```c
sword OCICollAssign ( OCIEnv              *env,
                         OCIError            *err,
                         const OCIColl       *rhs,
                         OCIColl             *lhs );
```

Parameters

- **env (IN/OUT)**
The OCI environment handle initialized in object mode.

  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
The OCI 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 Functions
`OCIErrorGet()`, `OCICollAssignElem()`
OCICollAssignElem()  

Purpose

Assigns the given element value elem to the element at coll[index].

Syntax

```
sword OCICollAssignElem ( OCIEnv           *env,  
                         OCIError         *err,  
                         sb4              index,  
                         const void       *elem,  
                         const void       *elemind,  
                         OCIColl          *coll );
```

Parameters

- **env (IN/OUT)**
The OCI environment handle initialized in object mode.

  See Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
The OCI 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.

- **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 Functions

OCIErrorGet(), OCICollAssign()
OCI CollGetElem()

Purpose

Gets a pointer to the element at the given index.

Syntax

```c
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 Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err** (IN/OUT)
The OCI 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 19–3 describes for each collection element type what the corresponding element pointer type is. The element pointer is returned with the `elem` parameter of `OCICollGetElem()`.

### Table 19–3  Element Pointers

<table>
<thead>
<tr>
<th>Element Type</th>
<th>*elem Is Set to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle NUMBER (OCINumber)</td>
<td>OCINumber*</td>
</tr>
</tbody>
</table>
**Table 19–3  (Cont.) Element Pointers**

<table>
<thead>
<tr>
<th>Element Type</th>
<th>*elem Is Set to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date (OCIDate)</td>
<td>OCIDate*</td>
</tr>
<tr>
<td>Datetime (OCIDateTime)</td>
<td>OCIDateTime*</td>
</tr>
<tr>
<td>Interval (OCIInterval)</td>
<td>OCIInterval*</td>
</tr>
<tr>
<td>Variable-length string (OCIString*)</td>
<td>OCISTring**</td>
</tr>
<tr>
<td>Variable-length raw (OCIRaw*)</td>
<td>OCIRaw**</td>
</tr>
<tr>
<td>Object reference (OCIRef*)</td>
<td>OCIRef**</td>
</tr>
<tr>
<td>Lob locator (OCILobLocator*)</td>
<td>OCILobLocator**</td>
</tr>
<tr>
<td>Object type (such as person)</td>
<td>person**</td>
</tr>
</tbody>
</table>

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.

**Example 19–1** shows how this can be accomplished with the `OCIRefAssign()` function.

**Example 19–1  Assigning a New Reference to the Pointer to the Collection Element**

```c
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*`. **Example 19–2** shows the prototype of the `OCINumberAssign()` call.

**Example 19–2  Prototype of OCINumberAssign() Call**

```c
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 Functions**

`OCIErrorGet()`, `OCICollAssignElem()`
OCI Coll Get Elem Array()

Purpose

Gets an array of elements from a collection when given a starting index.

Syntax

```c
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 Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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.

Returns

 Optionally, this function also returns the address of the element's NULL indicator information.
Related Functions

OCIErrorGet(), OCICollGetElem()
**OCI Coll Is Locator()**

**Purpose**
Indicates whether a collection is locator-based or not.

**Syntax**
```c
sword OCICollIsLocator ( OCIEnv *env,
    OCIError *err,
    const OCIColl *coll,
    boolean *result );
```

**Parameters**
- **env** (IN)
The OCI environment handle initialized in object mode.
  
  See Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err** (IN/OUT)
The OCI 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 Functions**
- `OCIErrorGet()`
OCICollMax()

Purpose

Gets the maximum size in number of elements of the given collection.

Syntax

```c
sb4 OCICollMax ( OCIEnv *env,
    const OCIColl *coll );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **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.

Returns

The upper bound of the given collection.

Related Functions

OCIErrorGet(), OCICollSize()
OCI CollSize()

Purpose

Gets the current size in number of elements of the given collection.

Syntax

sword OCI CollSize ( OCIEnv *env, OCIError *err, const OCIColl *coll, sb4 *size );

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

See Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

err (IN/OUT)
The OCI 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:

OCI CollSize(...);
// assume 'size' returned is equal to 5
OCITableDelete(...);  // delete one element
OCI CollSize(...);
// '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 Functions**

`OCIErrorGet()`, `OCICollMax()`
OCI CollTrim()

Purpose
Trims the given number of elements from the end of the collection.

Syntax
```c
sword OCICollTrim ( OCIEnv *env, OCIError *err, sb4 trim_num, OCIColl *coll );
```

Parameters
- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  See Also: "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 Functions
OCIErrorGet(), OCICollSize()
OCIIterCreate()

Purpose

Creates an iterator to scan the elements or the collection.

Syntax

```c
sword OCIIterCreate ( OCIEnv *env,
    OCIError *err,
    const OCIColl *coll,
    OCIIter **itr );
```

Parameters

- **env (IN/OUT)**
The OCI environment handle initialized in object mode.

  See Also: "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
The OCI 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 Oracle8i 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.

Returns

This function returns an error if any of the input parameters is `NULL`.

Related Functions

- `OCIErrorGet()`, `OCIIterDelete()`
OCIiterDelete()

Purpose

Deletes a collection iterator.

Syntax

```c
sword OCIiterDelete ( OCIEnv *env,
          OCIError *err,
          OCIIter **itr );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

- **err (IN/OUT)**
  The OCI 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()`.

Returns

This function returns an error if any of the input parameters is `NULL`.

Related Functions

- `OCIErrorGet()`, `OCIIterCreate()`
OCIIterGetCurrent()

Purpose

Gets a pointer to the current iterator collection element.

Syntax

```c
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 Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err** (IN/OUT)
The OCI 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.

- **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.

Returns

This function returns an error if any input parameter is NULL.

Related Functions

OCIErrorGet(), OCIIterNext(), OCIIterPrev()
OCIIterInit()

Purpose

Initializes an iterator to scan a collection.

Syntax

```c
sword OCIIterInit ( OCIEnv *env, OCIError *err, const OCIColl *coll, OCIIter *itr );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

- **err (IN/OUT)**
  The OCI 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 Oracle8i 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.

Returns

Returns an error if any input parameter is `NULL`.

Related Functions

- `OCIErrorGet()`
OCIterNext()

Purpose

Gets a pointer to the next iterator collection element.

Syntax

```c
sword OCIterNext ( 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 Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
The OCI 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.

- **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)**
TURE 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.

Returns

This function returns an error if any input parameter is NULL.
Related Functions

OCIErrorGet(), OCIIterGetCurrent(), OCIIterPrev()
OClIterPrev()

Purpose

Gets a pointer to the previous iterator collection element.

Syntax

```c
sword OClIterPrev ( OCIEnv *env,
                   OCIError *err,
                   OClIter *itr,
                   void **elem,
                   void **elemind,
                   boolean *boc );
```

Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode.

See Also: "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

**err (IN/OUT)**
The OCI 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.

**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.

Returns

This function returns an error if any input parameter is NULL.
Related Functions

OCIErrorGet(), OCIIterGetCurrent(), OCIIterNext()
OCI Date, Datetime, and Interval Functions

Table 19–4 describes the OCI date and interval functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIDateAddDays()&quot; on page 19-26</td>
<td>Add or subtract days</td>
</tr>
<tr>
<td>&quot;OCIDateAddMonths()&quot; on page 19-27</td>
<td>Add or subtract months</td>
</tr>
<tr>
<td>&quot;OCIDateAssign()&quot; on page 19-28</td>
<td>Assign date</td>
</tr>
<tr>
<td>&quot;OCIDateCheck()&quot; on page 19-29</td>
<td>Check if the given date is valid</td>
</tr>
<tr>
<td>&quot;OCIDateCompare()&quot; on page 19-31</td>
<td>Compare dates</td>
</tr>
<tr>
<td>&quot;OCIDateDaysBetween()&quot; on page 19-32</td>
<td>Get number of days between two dates</td>
</tr>
<tr>
<td>&quot;OCIDateFromText()&quot; on page 19-33</td>
<td>Convert string to date</td>
</tr>
<tr>
<td>&quot;OCIDateGetDate()&quot; on page 19-35</td>
<td>Get the date portion of a date</td>
</tr>
<tr>
<td>&quot;OCIDateGetTime()&quot; on page 19-36</td>
<td>Get the time portion of a date</td>
</tr>
<tr>
<td>&quot;OCIDateLastDay()&quot; on page 19-37</td>
<td>Get date of last day of month</td>
</tr>
<tr>
<td>&quot;OCIDateNextDay()&quot; on page 19-38</td>
<td>Get date of next day</td>
</tr>
<tr>
<td>&quot;OCIDateSetDate()&quot; on page 19-39</td>
<td>Set the date portion of a date</td>
</tr>
<tr>
<td>&quot;OCIDateSetTime()&quot; on page 19-40</td>
<td>Set the time portion of a date</td>
</tr>
<tr>
<td>&quot;OCIDateSysDate()&quot; on page 19-41</td>
<td>Get the current system date and time</td>
</tr>
<tr>
<td>&quot;OCIDateTimeAssign()&quot; on page 19-42</td>
<td>Perform a datetime assignment</td>
</tr>
<tr>
<td>&quot;OCIDateTimeCheck()&quot; on page 19-43</td>
<td>Check if the given date is valid</td>
</tr>
<tr>
<td>&quot;OCIDateTimeCompare()&quot; on page 19-45</td>
<td>Compare two datetime values</td>
</tr>
<tr>
<td>&quot;OCIDateTimeConstruct()&quot; on page 19-46</td>
<td>Construct a datetime descriptor</td>
</tr>
<tr>
<td>&quot;OCIDateTimeConvert()&quot; on page 19-48</td>
<td>Convert one datetime type to another</td>
</tr>
<tr>
<td>&quot;OCIDateTimeFromArray()&quot; on page 19-49</td>
<td>Convert an array of size OCI_DT_ARRAYLEN to an OCIDateTime descriptor</td>
</tr>
<tr>
<td>&quot;OCIDateTimeFromDate()&quot; on page 19-50</td>
<td>Convert the given string to Oracle datetime type in the OCIDateTime descriptor, according to the specified format</td>
</tr>
<tr>
<td>&quot;OCIDateTimeGetDate()&quot; on page 19-52</td>
<td>Get the date (year, month, day) portion of a datetime value</td>
</tr>
<tr>
<td>&quot;OCIDateTimeGetTime()&quot; on page 19-53</td>
<td>Get the time (hour, minute, second, fractional second) of a datetime value</td>
</tr>
<tr>
<td>&quot;OCIDateTimeGetTimeZoneName()&quot; on page 19-54</td>
<td>Get the time zone name portion of a datetime value</td>
</tr>
<tr>
<td>&quot;OCIDateTimeGetTimeZoneOffset()&quot; on page 19-55</td>
<td>Get the time zone (hour, minute) portion of a datetime value</td>
</tr>
<tr>
<td>&quot;OCIDateTimeIntervalAdd()&quot; on page 19-56</td>
<td>Add an interval to a datetime to produce a resulting datetime</td>
</tr>
<tr>
<td>&quot;OCIDateTimeIntervalSub()&quot; on page 19-57</td>
<td>Subtract an interval from a datetime and store the result in a datetime</td>
</tr>
<tr>
<td>Function</td>
<td>Purpose</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>&quot;OCIDateSubtract()&quot; on page 19-58</td>
<td>Take two datetimes as input and store their difference in an interval</td>
</tr>
<tr>
<td>&quot;OCIDateTimeSysTimeStamp()&quot; on page 19-59</td>
<td>Get the system current date and time as a time stamp with time zone</td>
</tr>
<tr>
<td>&quot;OCIDateTimeToArray()&quot; on page 19-60</td>
<td>Convert an OCIDateTime descriptor to an array</td>
</tr>
<tr>
<td>&quot;OCIDateTimeToText()&quot; on page 19-61</td>
<td>Convert the given date to a string according to the specified format</td>
</tr>
<tr>
<td>&quot;OCIDateToText()&quot; on page 19-63</td>
<td>Convert date to string</td>
</tr>
<tr>
<td>&quot;OCIIntervalDivide()&quot; on page 19-65</td>
<td>Divide an interval by an Oracle NUMBER to produce an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalFromNumber()&quot; on page 19-67</td>
<td>Add two intervals to produce a resulting interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalAssign()&quot; on page 19-68</td>
<td>Copy one interval to another</td>
</tr>
<tr>
<td>&quot;OCIIntervalCheck()&quot; on page 19-69</td>
<td>Check the validity of an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalCompare()&quot; on page 19-71</td>
<td>Compare two intervals</td>
</tr>
<tr>
<td>&quot;OCIIntervalFromTZ()&quot; on page 19-72</td>
<td>Divide an interval by an Oracle NUMBER to produce an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalFromText()&quot; on page 19-73</td>
<td>Convert an Oracle NUMBER to an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalGetDaySecond()&quot; on page 19-74</td>
<td>When given an interval string, return the interval represented by the string</td>
</tr>
<tr>
<td>&quot;OCIIntervalGetYearMonth()&quot; on page 19-75</td>
<td>Return an OCI_DTYPE_INTERVAL_DS</td>
</tr>
<tr>
<td>&quot;OCIIntervalMultiply()&quot; on page 19-76</td>
<td>Get values of day, hour, minute, and second from an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalSetDaySecond()&quot; on page 19-77</td>
<td>Get year and month from an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalSetYearMonth()&quot; on page 19-78</td>
<td>Multiply an interval by an Oracle NUMBER to produce an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalSetDaySecond()&quot; on page 19-79</td>
<td>Set day, hour, minute, and second in an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalSetYearMonth()&quot; on page 19-80</td>
<td>Set year and month in an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalSubtract()&quot; on page 19-81</td>
<td>Subtract two intervals and stores the result in an interval</td>
</tr>
<tr>
<td>&quot;OCIIntervalToNumber()&quot; on page 19-82</td>
<td>Convert an interval to an Oracle NUMBER</td>
</tr>
<tr>
<td>&quot;OCIIntervalToText()&quot; on page 19-83</td>
<td>When given an interval, produce a string representing the interval</td>
</tr>
</tbody>
</table>
**OCIDateAddDays()**

**Purpose**

Adds or subtracts days from a given date.

**Syntax**

```c
sword OCIDateAddDays ( OCIError *err,
                        const OCIDate *date,
                        sb4 num_days,
                        OCIDate *result );
```

**Parameters**

- **err (IN/OUT)**
  The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

- **date (IN)**
  The given date from which to add or subtract.

- **num_days (IN)**
  Number of days to be added or subtracted. A negative value is subtracted.

- **result (IN/OUT)**
  Result of adding days to, or subtracting days from, `date`.

**Returns**

This function returns an error if an invalid date is passed to it.

**Related Functions**

- `OCIErrorGet()`
- `OCIDateAddMonths()`
OCIDateAddMonths()

Purpose

Adds or subtracts months from a given date.

Syntax

```c
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 Functions

`OCIErrorGet()`, `OCIDateAddDays()`
OCIDateAssign()

Purpose

Performs a date assignment.

Syntax

```c
sword OCIDateAssign ( OCIError *err,
                      const OCIDate *from,
                      OCIDate *to );
```

Parameters

- **err (IN/OUT)**
  The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

- **from (IN)**
  Date to be assigned.

- **to (OUT)**
  Target of assignment.

Comments

This function assigns a value from one `OCIDate` variable to another.

Related Functions

- `OCIErrorGet()`, `OCIDateCheck()`
OCIDateCheck()

Purpose

Checks if the given date is valid.

Syntax

```c
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 19–5.

### Table 19–5   Error Bits Returned by the valid Parameter for OCIDateCheck()

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Bit Number</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_DATE_INVALID_DAY</td>
<td>0x1</td>
<td>Bad day</td>
</tr>
<tr>
<td>OCI_DATE_DAY_BELOW_VALID</td>
<td>0x2</td>
<td>Bad day low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DATE_INVALID_MONTH</td>
<td>0x4</td>
<td>Bad month</td>
</tr>
<tr>
<td>OCI_DATE_MONTH_BELOW_VALID</td>
<td>0x8</td>
<td>Bad month low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DATE_INVALID_YEAR</td>
<td>0x10</td>
<td>Bad year</td>
</tr>
<tr>
<td>OCI_DATE_YEAR_BELOW_VALID</td>
<td>0x20</td>
<td>Bad year low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DATE_INVALID_HOUR</td>
<td>0x40</td>
<td>Bad hour</td>
</tr>
<tr>
<td>OCI_DATE_HOUR_BELOW_VALID</td>
<td>0x80</td>
<td>Bad hour low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DATE_INVALID_MINUTE</td>
<td>0x100</td>
<td>Bad minute</td>
</tr>
<tr>
<td>OCI_DATE_MINUTE_BELOW_VALID</td>
<td>0x200</td>
<td>Bad minute low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DATE_INVALID_SECOND</td>
<td>0x400</td>
<td>Bad second</td>
</tr>
<tr>
<td>OCI_DATE_SECOND_BELOW_VALID</td>
<td>0x800</td>
<td>Bad second low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DATE_DAY_MISSING_FROM_1582</td>
<td>0x1000</td>
<td>Day is one of those missing from 1582</td>
</tr>
<tr>
<td>OCI_DATE_YEAR_ZERO</td>
<td>0x2000</td>
<td>Year may not equal zero</td>
</tr>
<tr>
<td>OCI_DATE_INVALID_FORMAT</td>
<td>0x8000</td>
<td>Bad date format input</td>
</tr>
</tbody>
</table>

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:

```c
OCI_DATE_INVALID_DAY | OCI_DATE_DAY_BELOW_VALID | OCI_DATE_INVALID_MONTH | OCI_DATE_INVALID_HOUR | OCI_DATE_INVALID_MINUTE.
```
Returns

This function returns an error if date or valid pointer is NULL.

Related Functions

OCIErrorGet(), OCIDateCompare()
OCIDateCompare()

Purpose

Compares two dates.

Syntax

```c
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 19–6.

<table>
<thead>
<tr>
<th>Comparison Result</th>
<th>Output in result Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date1 &lt; date2</code></td>
<td>-1</td>
</tr>
<tr>
<td><code>date1 = date2</code></td>
<td>0</td>
</tr>
<tr>
<td><code>date1 &gt; date2</code></td>
<td>1</td>
</tr>
</tbody>
</table>

Returns

This function returns an error if an invalid date is passed to it.

Related Functions

- `OCIErrorGet()`, `OCIDateCheck()`
OCIDateDaysBetween()

Purpose

Gets the number of days between two dates.

Syntax

```c
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 Functions

- `OCIErrorGet()`, `OCIDateCheck()`
OCIDateFromText()

Purpose

Converts a character string to a date type according to the specified format.

Syntax

```c
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 Functions

OCIErrorGet(), OCIDateToText()
**OCIDateGetDate()**

**Purpose**

Gets the year, month, and day stored in an Oracle date.

**Syntax**

```c
void OCIDateGetDate ( const OCIDate    *date,
                     sb2              *year,
                     ubl              *month,
                     ubl              *day );
```

**Parameters**

- **date (IN)**
  Oracle date whose year, month, day data is retrieved.

- **year (OUT)**
  Year value returned.

- **month (OUT)**
  Month value returned.

- **day (OUT)**
  Day value returned.

**Comments**

None.

**Related Functions**

- OCIDateSetDate(), OCIDateGetTime()
OCIDateGetTime()

Purpose

Gets the time stored in an Oracle date.

Syntax

```c
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 Functions

OCIDateSetTime(), OCIDateGetDate()
OCIDateLastDay()

Purpose

Gets the date of the last day of the month in a specified date.

Syntax

```c
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 Functions

- `OCIErrorGet()`, `OCIDateGetDate()`
OCIDateNextDay()

Purpose

Gets the date of the next day of the week after a given date.

Syntax

```c
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

Example 19–3 shows how to get the date of the next Monday after April 18, 1996 (a Thursday).

```c
Example 19–3 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 Functions

`OCIErrorGet()`, `OCIDateGetDate()`
OCIDateSetDate()

Purpose
Set the values in an Oracle date.

Syntax
void OCIDateSetDate ( OCIDate    *date,
                     sb2        year,
                     ubl        month,
                     ubl        day );

Parameters

date (OUT)
Oracle date whose time data is set.

year (IN)
Year value to be set.

month (IN)
Month value to be set.

day (IN)
Day value to be set.

Comments
None.

Related Functions
OCIDateGetDate()
OCIDateSetTime()

Purpose

Sets the time information in an Oracle date.

Syntax

```c
void OCIDateSetTime ( OCIDate    *date,
                        ub1        hour,
                        ub1        min,
                        ub1        sec );
```

Parameters

- **date (OUT)**
  Oracle date whose time data is set.

- **hour (IN)**
  Hour value to be set.

- **min (IN)**
  Minute value to be set.

- **sec (IN)**
  Second value to be set.

Comments

None.

Related Functions

OCIDateGetTime()
**OCIDateSysDate()**

**Purpose**

Gets the current system date and time of the client.

**Syntax**

```c
sword OCIDateSysDate ( OCIError *err,
                         OCIDate       *sys_date );
```

**Parameters**

- **err (IN/OUT)**
  The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

- **sys_date (OUT)**
  Current system date and time of the client.

**Comments**

None.

**Related Functions**

`OCIErrorGet()`
**OCIDateTimeAssign()**

**Purpose**

Performs a datetime assignment.

**Syntax**

```c
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 `NLSCALENDAR`; 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 Functions**

- `OCIDateTimeCheck()`, `OCIDateTimeConstruct()`
**OCIDateTimeCheck()**

**Purpose**

Checks if the given date is valid.

**Syntax**

```c
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 NLSCALENDAR, 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 19–7.

**Table 19–7  Error Bits Returned by the valid Parameter for OCIDateTimeCheck()**

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Bit Number</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_DT_INVALID_DAY</td>
<td>0x1</td>
<td>Bad day</td>
</tr>
<tr>
<td>OCI_DT_DAY BELOW VALID</td>
<td>0x2</td>
<td>Bad day low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DT_INVALID_MONTH</td>
<td>0x4</td>
<td>Bad month</td>
</tr>
<tr>
<td>OCI_DT_MONTH BELOW VALID</td>
<td>0x8</td>
<td>Bad month low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DT_INVALID_YEAR</td>
<td>0x10</td>
<td>Bad year</td>
</tr>
<tr>
<td>OCI_DT_YEAR BELOW VALID</td>
<td>0x20</td>
<td>Bad year low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DT_INVALID_HOUR</td>
<td>0x40</td>
<td>Bad hour</td>
</tr>
<tr>
<td>OCI_DT_HOUR BELOW VALID</td>
<td>0x80</td>
<td>Bad hour low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DT_INVALID_MINUTE</td>
<td>0x100</td>
<td>Bad minute</td>
</tr>
<tr>
<td>OCI_DT_MINUTE BELOW VALID</td>
<td>0x200</td>
<td>Bad minute low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DT_INVALID_SECOND</td>
<td>0x400</td>
<td>Bad second</td>
</tr>
<tr>
<td>OCI_DT_SECOND BELOW VALID</td>
<td>0x800</td>
<td>Bad second low/high bit (1=low)</td>
</tr>
<tr>
<td>OCI_DT_DAY_MISSING_FROM_1582</td>
<td>0x1000</td>
<td>Day is one of those missing from 1582</td>
</tr>
<tr>
<td>OCI_DT_YEAR ZERO</td>
<td>0x2000</td>
<td>Year may not equal zero</td>
</tr>
<tr>
<td>OCI_DT_INVALID_TIMEZONE</td>
<td>0x4000</td>
<td>Bad time zone</td>
</tr>
</tbody>
</table>
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:

\[
\text{OCI_DT_INVALID_DAY} | \text{OCI_DT_DAY_BELOW_VALID} | \\
\text{OCI_DT_INVALID_HOUR} | \text{OCI_DT_INVALID_MINUTE}.
\]

**Returns**

OCI_SUCCESS; OCI_INVALID_HANDLE, if err is a NULL pointer; OCI_ERROR, if date or valid is a NULL pointer.

**Related Functions**

OCIDateTimeAssign()
OCIDateTimeCompare()

Purpose

Compares two datetime values.

Syntax

```c
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 19–8.

### Table 19–8 Comparison Results Returned by the result Parameter for OCIDateTimeCompare()

<table>
<thead>
<tr>
<th>Comparison Result</th>
<th>Output in result Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date1 &lt; date2</code></td>
<td>-1</td>
</tr>
<tr>
<td><code>date1 = date2</code></td>
<td>0</td>
</tr>
<tr>
<td><code>date1 &gt; date2</code></td>
<td>1</td>
</tr>
</tbody>
</table>

Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`, if `err` is a NULL pointer; `OCI_ERROR`, if an invalid date is used or if the input date arguments are not of mutually comparable types.

Related Functions

`OCIDateTimeConstruct()`
OCIDateTimeConstruct()

Purpose

Constructs a datetime descriptor.

Syntax

```c
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).
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 Functions**

OCIDateTimeAssign(), OCIDateTimeConvert()
OCIDateTimeConvert()  

Purpose  
Converts one datetime type to another.

Syntax  
```c
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 Functions  
- `OCIDateTimeCheck()`
OCIDateFromArray()

Purpose

Converts an array containing a date to an OCIDateTime descriptor.

Syntax

```c
sword OCIDateFromArray ( void *hndl, OCIError *err,
                          const ub1 *inarray,
                          ub4 *len
                          ub1 type,
                          OCIDateTime *datetime,
                          const OCIInterval *refzt,
                          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.
- **refzt** (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 Functions

- `OCIDateTimeFromText()`, `OCIDateTimeToArray()`
OCIDateTimeFromText()

Purpose

Converts the given string to an Oracle datetime type in the OCIDateTime descriptor, according to the specified format.

Syntax

```c
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 NLSCALENDAR; 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 Functions

`OCIDateTimeToText()`, `OCIDateTimeFromArray()`.
OCIDateTimeGetDate()

Purpose

 Gets the date (year, month, day) portion of a datetime value.

Syntax

 void 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, if the input type is SQLT_TIME or OCI_TIME_TZ.

Related Functions

 OCIDateTimeGetTime()
OCI DateTime Get Time

Purpose

Gets the time (hour, min, second, fractional second) of a datetime value.

Syntax

```c
void 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`;
- `OCI_ERROR`, if datetime does not contain time (`SQLT_DATE`).

Related Functions

- `OCIDateTimeGetDate()`
OCIDateTimeGetTimeZoneName()

Purpose

Gets the time zone name portion of a datetime value.

Syntax

```c
void 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`;
- `OCI_ERROR`, if `datetime` does not contain a time zone (SQLT_DATE, SQLT_TIMESTAMP).

Related Functions

- `OCIDateTimeGetDate()`, `OCIDateTimeGetTimeZoneOffset()`
OCI Date, Datetime, and Interval Functions

OCIDateTimeGetTimeZoneOffset()

Purpose

Gets the time zone (hour, minute) portion of a datetime value.

Syntax

```c
void OCIDateTimeGetTimeZoneOffset ( void               *hndl,
                                      OCIError           *err,
                                      const OCIDateTime  *datetime,
                                      sbl                *hour,
                                      sbl                *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 Functions

- `OCIDateTimeGetDate()`, `OCIDateTimeGetTimeZoneName()`
OCIDateTimeIntervalAdd()

Purpose

Adds an interval to a datetime to produce a resulting datetime.

Syntax

```c
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 Functions

`OCIDateTimeIntervalSub()`
OCI Date, Datetime, and Interval Functions

OCIDateTimeIntervalSub()

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 Functions
OCIDateTimeIntervalAdd()
OCIDateTimeSubtract()

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 Functions

OCIDateTimeCompare()
OCI Date, Datetime, and Interval Functions

OCI DateTimeSysTimeStamp()

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 Functions

OCIDateSysDate()
OCIDateTimeToArray()

Purpose

Converts an OCIDateTime descriptor to an array.

Syntax

```c
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 Functions

OCIDateTimeToText(), OCIDateTimeFromArray()
OCIDateTImeToText()

Purpose

Converts the given date to a string according to the specified format.

Syntax

```c
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 NLSCALENDAR; 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 Functions

OCIDateTimeFromText()
OCIDateToText()

**Purpose**

Converts a date type to a character string.

**Syntax**

```c
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 'y' causes an error.

Related Functions

OCIErrorGet(), OCIDateFromText()
OCI Date, Datetime, and Interval Functions

OCI DateZoneToZone()

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 Functions

OCIErrorGet(), OCIDateCheck()
OCIIntervalAdd()

Purpose

Adds two intervals to produce a resulting interval.

Syntax

```c
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 `SB4_MAXVAL`.
  - The resulting year is less than `SB4_MINVAL`.

Related Functions

- `OCIIntervalSubtract()`
OCIIntervalAssign()

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 Functions
OCIIntervalCompare()
OCIIntervalCheck()

Purpose

Checks the validity of an interval.

Syntax

```c
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 19–9.

Table 19–9  Error Bits Returned by the valid Parameter for OCIIntervalCheck()

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Bit Number</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIINTER_INVALID_DAY</td>
<td>0x1</td>
<td>Bad day</td>
</tr>
<tr>
<td>OCIINTER_DAY_BELOW_VALID</td>
<td>0x2</td>
<td>Bad day low/high bit (1=low)</td>
</tr>
<tr>
<td>OCIINTER_INVALID_MONTH</td>
<td>0x4</td>
<td>Bad month</td>
</tr>
<tr>
<td>OCIINTER_MONTH_BELOW_VALID</td>
<td>0x8</td>
<td>Bad month low/high bit (1=low)</td>
</tr>
<tr>
<td>OCIINTER_INVALID_YEAR</td>
<td>0x10</td>
<td>Bad year</td>
</tr>
<tr>
<td>OCIINTER_YEAR_BELOW_VALID</td>
<td>0x20</td>
<td>Bad year low/high bit (1=low)</td>
</tr>
<tr>
<td>OCIINTER_INVALID_HOUR</td>
<td>0x40</td>
<td>Bad hour</td>
</tr>
<tr>
<td>OCIINTER_HOUR_BELOW_VALID</td>
<td>0x80</td>
<td>Bad hour low/high bit (1=low)</td>
</tr>
<tr>
<td>OCIINTER_INVALID_MINUTE</td>
<td>0x100</td>
<td>Bad minute</td>
</tr>
<tr>
<td>OCIINTER_MINUTE_BELOW_VALID</td>
<td>0x200</td>
<td>Bad minute low/high bit (1=low)</td>
</tr>
<tr>
<td>OCIINTER_INVALID_SECOND</td>
<td>0x400</td>
<td>Bad second</td>
</tr>
<tr>
<td>OCIINTER_SECOND_BELOW_VALID</td>
<td>0x800</td>
<td>Bad second low/high bit (1=low)</td>
</tr>
<tr>
<td>OCIINTER_INVALID_FRACSEC</td>
<td>0x1000</td>
<td>Bad fractional second</td>
</tr>
<tr>
<td>OCIINTER_FRACSEC_BELOW_VALID</td>
<td>0x2000</td>
<td>Bad fractional second low/high bit (1=low)</td>
</tr>
</tbody>
</table>
Returns

OCI_SUCCESS; OCI_INVALID_HANDLE, if err is a NULL pointer; or OCI_ERROR, on error.

Related Functions

OCIIntervalCompare()
OCIIntervalCompare()

Purpose

Compares two intervals.

Syntax

```c
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 19–10.

Table 19–10  Comparison Results Returned by the result Parameter for OCIIntervalCompare()

<table>
<thead>
<tr>
<th>Comparison Result</th>
<th>Output in result Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>inter1 &lt; inter2</td>
<td>-1</td>
</tr>
<tr>
<td>inter1 = inter2</td>
<td>0</td>
</tr>
<tr>
<td>inter1 &gt; inter2</td>
<td>1</td>
</tr>
</tbody>
</table>

Returns

`OCI_SUCCESS`; `OCI_INVALID_HANDLE`, if `err` is a NULL pointer; or `OCI_ERROR`, if the input values are not mutually comparable.

Related Functions

- **OCIIntervalAssign()**
OCIIntervalDivide()

Purpose

Divides an interval by an Oracle NUMBER to produce an interval.

Syntax

```c
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 Functions

OCIIntervalMultiply()
OCIIntervalFromNumber()

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 Functions
OCIIntervalToNumber()
OCIIntervalFromText()

**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**

```c
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 Functions**

**OCIIntervalToText()**
OCIIntervalFromTZ()

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,
OCIErr            *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 Functions

OCIIntervalFromText()
OCIIntervalGetDaySecond()

Purpose

Gets values of day, hour, minute, and second from an interval.

Syntax

```c
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 fractional seconds.

- **interval (IN)**
  The input interval.

Returns

`OCI_SUCCESS`; or `OCI_INVALID_HANDLE`, if `err` is a NULL pointer.

Related Functions

- `OCIIntervalSetDaySecond()`
OCIIntervalGetYearMonth()

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.

er (IN/OUT)
The OCI 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 Functions

OCIIntervalSetYearMonth()
OCIIntervalMultiply()

Purpose

Multiplies an interval by an Oracle NUMBER to produce an interval.

Syntax

```c
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 Functions

- `OCIIntervalDivide()`
OCIIntervalSetDaySecond()

Purpose
Sets day, hour, minute, and second in an interval.

Syntax
```c
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 fractional seconds.

- **result (OUT)**
The resulting interval.

Returns
OCI_SUCCESS; or OCI_INVALID_HANDLE, if err is a NULL pointer.

Related Functions
OCIIntervalGetDaySecond()
OCIIntervalSetYearMonth()

Purpose

Sets year and month in an interval.

Syntax

```c
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 Functions

- `OCIIntervalGetYearMonth()`
OCIIntervalSubtract()

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 Functions

OCIIntervalAdd()
OCIIntervalToNumber()

Purpose

Converts an interval to an Oracle NUMBER.

Syntax

```c
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`;
- `OCI_INVALID_HANDLE`, if `err` is a NULL pointer.

Related Functions

- `OCIIntervalFromNumber()`
OCIIntervalToText()

**Purpose**
When given an interval, produces a string representing the interval.

**Syntax**
```c
sword OCIIntervalToText ( void               *hndl,
OCIErrror           *err,
const OCIInterval   *interval,
ub1                lfprec,
ub1                fsprec,
OraText            *buffer,
size_t             buflen,
size_t             *resultlen );
```

**Parameters**
- **hndl (IN)**
The OCI user session handle or the environment handle.
- **err (IN/OUT)**
The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.
- **interval (IN)**
Interval to be converted.
- **lfprec (IN)**
Leading field precision. (The number of digits used to represent the leading field.)
- **fsprec (IN)**
Fractional second precision of the interval (the number of digits used to represent the fractional seconds).
- **buffer (OUT)**
Buffer to hold the result.
- **buflen (IN)**
The length of `buffer`.
- **resultlen (OUT)**
The length of the result placed into `buffer`.

**Comments**
The interval literal is output as ‘year’ or ‘[year-]month’ for `INTERVAL YEAR TO MONTH` intervals and as ‘seconds’ or ‘minutes[seconds]’ or ‘hours[minutes][seconds]’ or ‘days[ hours[minutes][seconds]]’ for `INTERVAL DAY TO SECOND` intervals (where optional fields are surrounded by brackets).

**Returns**
- `OCI_SUCCESS`;
- `OCI_INVALID_HANDLE`, if `err` is a NULL pointer;
- `OCI_ERROR`, if the buffer is not large enough to hold the result.
Related Functions

OCIIntervalFromText()
OCI NUMBER Functions

Table 19–11 describes the OCI NUMBER functions that are described in this section.

See Also: "OCINumber Examples" on page 12-10

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCINumberAbs()&quot; on page 19-87</td>
<td>Compute the absolute value</td>
</tr>
<tr>
<td>&quot;OCINumberAdd()&quot; on page 19-88</td>
<td>Add NUMBERS</td>
</tr>
<tr>
<td>&quot;OCINumberArcCos()&quot; on page 19-89</td>
<td>Compute the arc cosine</td>
</tr>
<tr>
<td>&quot;OCINumberArcSin()&quot; on page 19-90</td>
<td>Compute the arc sine</td>
</tr>
<tr>
<td>&quot;OCINumberArcTan()&quot; on page 19-91</td>
<td>Compute the arc tangent</td>
</tr>
<tr>
<td>&quot;OCINumberArcTan2()&quot; on page 19-92</td>
<td>Compute the arc tangent of two NUMBERS</td>
</tr>
<tr>
<td>&quot;OCINumberAssign()&quot; on page 19-93</td>
<td>Assign one NUMBER to another</td>
</tr>
<tr>
<td>&quot;OCINumberCeil()&quot; on page 19-94</td>
<td>Compute the ceiling of NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberCmp()&quot; on page 19-95</td>
<td>Compare NUMBERS</td>
</tr>
<tr>
<td>&quot;OCINumberCos()&quot; on page 19-96</td>
<td>Compute the cosine</td>
</tr>
<tr>
<td>&quot;OCINumberDec()&quot; on page 19-97</td>
<td>Decrement a NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberDiv()&quot; on page 19-98</td>
<td>Divide two NUMBERS</td>
</tr>
<tr>
<td>&quot;OCINumberExp()&quot; on page 19-99</td>
<td>Raise e to the specified Oracle NUMBER power</td>
</tr>
<tr>
<td>&quot;OCINumberFloor()&quot; on page 19-100</td>
<td>Compute the floor value of a NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberFromInt()&quot; on page 19-101</td>
<td>Convert an integer to an Oracle NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberFromReal()&quot; on page 19-102</td>
<td>Convert a real type to an Oracle NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberFromText()&quot; on page 19-103</td>
<td>Convert a string to an Oracle NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberHypCos()&quot; on page 19-105</td>
<td>Compute the hyperbolic cosine</td>
</tr>
<tr>
<td>&quot;OCINumberHypSin()&quot; on page 19-106</td>
<td>Compute the hyperbolic sine</td>
</tr>
<tr>
<td>&quot;OCINumberHypTan()&quot; on page 19-107</td>
<td>Compute the hyperbolic tangent</td>
</tr>
<tr>
<td>&quot;OCINumberInc()&quot; on page 19-108</td>
<td>Increment an Oracle NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberIntPower()&quot; on page 19-109</td>
<td>Raise a given base to an integer power</td>
</tr>
<tr>
<td>&quot;OCINumberIsInt()&quot; on page 19-110</td>
<td>Test if a NUMBER is an integer</td>
</tr>
<tr>
<td>&quot;OCINumberIsZero()&quot; on page 19-111</td>
<td>Test if a NUMBER is zero</td>
</tr>
<tr>
<td>&quot;OCINumberLn()&quot; on page 19-112</td>
<td>Compute the natural logarithm</td>
</tr>
<tr>
<td>&quot;OCINumberLog()&quot; on page 19-113</td>
<td>Compute the logarithm to an arbitrary base</td>
</tr>
<tr>
<td>&quot;OCINumberMod()&quot; on page 19-114</td>
<td>Gets the modulus (remainder) of the division of two Oracle NUMBERS</td>
</tr>
<tr>
<td>&quot;OCINumberMul()&quot; on page 19-115</td>
<td>Multiply two Oracle NUMBERS</td>
</tr>
<tr>
<td>&quot;OCINumberNeg()&quot; on page 19-116</td>
<td>Negates an Oracle NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberPower()&quot; on page 19-117</td>
<td>Raises a given base to a given exponent</td>
</tr>
<tr>
<td>&quot;OCINumberPrec()&quot; on page 19-118</td>
<td>Round a NUMBER to a specified number of decimal places</td>
</tr>
</tbody>
</table>
## OCI NUMBER Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCINumberRound()&quot; on page 19-119</td>
<td>Round an Oracle NUMBER to a specified decimal place</td>
</tr>
<tr>
<td>&quot;OCINumberSetPi()&quot; on page 19-120</td>
<td>Initialize a NUMBER to pi</td>
</tr>
<tr>
<td>&quot;OCINumberSetZero()&quot; on page 19-121</td>
<td>Initialize a NUMBER to zero</td>
</tr>
<tr>
<td>&quot;OCINumberShift()&quot; on page 19-122</td>
<td>Multiply by 10, shifting a specified number of decimal places</td>
</tr>
<tr>
<td>&quot;OCINumberSign()&quot; on page 19-123</td>
<td>Obtain the sign of an Oracle NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberSin()&quot; on page 19-124</td>
<td>Compute the sine</td>
</tr>
<tr>
<td>&quot;OCINumberSqrt()&quot; on page 19-125</td>
<td>Compute the square root of a NUMBER</td>
</tr>
<tr>
<td>&quot;OCINumberSub()&quot; on page 19-126</td>
<td>Subtract NUMBERS</td>
</tr>
<tr>
<td>&quot;OCINumberTan()&quot; on page 19-127</td>
<td>Compute the tangent</td>
</tr>
<tr>
<td>&quot;OCINumberToInt()&quot; on page 19-128</td>
<td>Convert an Oracle NUMBER to an integer</td>
</tr>
<tr>
<td>&quot;OCINumberToReal()&quot; on page 19-129</td>
<td>Convert an Oracle NUMBER to a real type</td>
</tr>
<tr>
<td>&quot;OCINumberToRealArray()&quot; on page 19-130</td>
<td>Convert an array of NUMBER to a real array.</td>
</tr>
<tr>
<td>&quot;OCINumberToText()&quot; on page 19-131</td>
<td>Convert an Oracle NUMBER to a string</td>
</tr>
<tr>
<td>&quot;OCINumberTrunc()&quot; on page 19-133</td>
<td>Truncate an Oracle NUMBER at a specified decimal place</td>
</tr>
</tbody>
</table>
OCINumberAbs()

Purpose

Computes the absolute value of an Oracle NUMBER.

Syntax

```c
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 Functions

- `OCIErrorGet()`
**OCINumberAdd()**

**Purpose**
Adds two Oracle NUMBERs together.

**Syntax**
```c
sword OCINumberAdd ( OCIError              *err,
                     const OCINumber       *number1,
                     const OCINumber       *number2,
                     OCINumber             *result );
```

**Parameters**

- **err (IN/OUT)**
The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

- **number1, number2 (IN)**
NUMBERs to be added.

- **result (OUT)**
Result of adding `number1` to `number2`.

**Returns**
This function returns an error if any of the NUMBER arguments is NULL.

**Related Functions**
- `OCIErrorGet()`, `OCINumberSub()`
OCI NUMBER Functions

OCI NUMBER ArcCos()

Purpose
Takes the arc cosine in radians of an Oracle NUMBER.

Syntax
sword OCI NUMBER ArcCos ( OCI Error *err, 
                     const OCI Number *number, 
                     OCI Number *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 OCI Error Get().

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 Functions
OCI Error Get(), OCI Number Cos()
**OCINumberArcSin()**

**Purpose**
Takes the arc sine in radians of an Oracle `NUMBER`.

**Syntax**
```c
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 Functions**
- `OCIErrorGet()`, `OCINumberSin()`
OCI Number ArcTan()

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().

tnumber (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 Functions

OCIErrorGet(), OCINumberTan()
OCINumberArcTan2()

Purpose

Takes the arc tangent of two Oracle NUMBERS.

Syntax

```c
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 OCIErrrorGet().

- **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 Functions

OCIErrrorGet(), OCINumberTan()
OCI NUMBER Functions

OCINumberAssign()

Purpose

Assigns one Oracle NUMBER to another Oracle NUMBER.

Syntax

sword OCINumberAssign ( OCIError              *err,
                       const OCINumber       *from,
                       OCINumber             *to );

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

from (IN)
NUMBER to be assigned.

to (OUT)
NUMBER copied into.

Comments

Assigns the NUMBER identified by from to the NUMBER identified by to.

Returns

This function returns an error if any of the NUMBER arguments is NULL.

Related Functions

OCIErrorGet(), OCINumberCmp()
OCINumberCeil()

Purpose

Computes the ceiling value of an Oracle NUMBER.

Syntax

```c
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 Functions

- `OCIErrorGet()`, `OCINumberFloor()`
OCINumberCmp()

Purpose

Compares two Oracle NUMBERS.

Syntax

```c
sword OCINumberCmp ( OCIError *err,
                   const OCINumber *number1,
                   const OCINumber *number2,
                   sword *result );
```

Parameters

- **err (IN/OUT)**
  The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

- **number1, number2 (IN)**
  NUMBERS to compare.

- **result (OUT)**
  Comparison result as specified in Table 19–12.

Table 19–12 Comparison Results Returned by the result Parameter for OCINumberCmp()

<table>
<thead>
<tr>
<th>Comparison Result</th>
<th>Output in result Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>number1 &lt; number2</td>
<td>negative</td>
</tr>
<tr>
<td>number1 = number2</td>
<td>0</td>
</tr>
<tr>
<td>number1 &gt; number2</td>
<td>positive</td>
</tr>
</tbody>
</table>

Returns

This function returns an error if any of the NUMBER arguments is NULL.

Related Functions

`OCIErrorGet()`, `OCINumberAssign()`
OCINumberCos()

Purpose

Computes the cosine in radians of an Oracle NUMBER.

Syntax

```c
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 Functions

`OCIErrorGet()`, `OCINumberArcCos()`
OCINumberDec()

Purpose
Decrement 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
Decrement 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 Functions
OCINumberInc()
OCINumberDiv()

Purpose

Divides two Oracle NUMBERs.

Syntax

```c
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 Functions

- `OCIErrorGet()`, `OCINumberMul()`
OCI NUMBER Functions

OCINumberExp()

Purpose

 Raises $e$ to the specified Oracle NUMBER power.

Syntax

```
sword OCINumberExp ( OCIError *err,
                    const OCINumber *number,
                    OCINumber *result );
```

Parameters

```
err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
This function raises $e$ to this Oracle NUMBER power.

result (OUT)
Output of exponentiation.
```

Returns

This function returns an error if any of the NUMBER arguments is NULL.

Related Functions

OCIErrorGet(), OCINumberLn()
OCINumberFloor()

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 Functions
OCIErrorGet(), OCINumberCeil()
OCI NUMBER Functions

OCINumberFromInt()

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 Functions

OCIErrorGet(), OCINumberToInt()
OCINumberFromReal()

Purpose

Converts a real (floating-point) type to an Oracle NUMBER.

Syntax

```c
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 Functions

- `OCIErrorGet()`, `OCINumberToReal()`
OCI NUMBER Functions

OCINumberFromText()

Purpose

Converts a character string to an Oracle NUMBER.

Syntax

```c
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.
Related Functions

OCIErrorGet(), OCINumberToText()
OCINumberHypCos()

Purpose
Computes the hyperbolic cosine of an Oracle NUMBER.

Syntax
sword OCINumberHypCos ( OCIError *err,
const OCINumber *number,
OCINumber *result );

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the cosine hyperbolic.

result (OUT)
Result of the cosine hyperbolic.

Returns
This function returns an error if either of the number arguments is NULL.

Caution: An Oracle NUMBER overflow causes an unpredictable result value.

Related Functions
OCIErrorGet(), OCINumberHypSin(), OCINumberHypTan()
OCINumberHypSin()

Purpose

Computes the hyperbolic sine of an Oracle NUMBER.

Syntax

```c
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`.

---

**Caution:** An Oracle `NUMBER` overflow causes an unpredictable result value.

---

Related Functions

- `OCIErrorGet()`, `OCINumberHypCos()`, `OCINumberHypTan()`
OCI NUMBER Functions

OCINumberHypTan()

Purpose
Computes the hyperbolic tangent of an Oracle NUMBER.

Syntax
sword OCINumberHypTan ( OCIError              *err,
                         const OCINumber       *number,
                         OCINumber             *result );

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the tangent hyperbolic.

result (OUT)
Result of the tangent hyperbolic.

Returns
This function returns an error if either of the NUMBER arguments is NULL.

Caution: An Oracle NUMBER overflow causes an unpredictable result value.

Related Functions
OCIErrorGet(), OCINumberHypCos(), OCINumberHypSin()
OCINumberInc()

Purpose

Increments an Oracle NUMBER.

Syntax

```c
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 Functions

- `OCINumberDec()`
OCI NUMBER Functions

OCINumberIntPower()

Purpose

 Raises a given base to a given integer power.

Syntax

sword OCINumberIntPower ( OCIError *err, 
                        const OCINumber *base, 
                        const sword exp, 
                        OCINumber *result );

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

base (IN)
Base of the exponentiation.

exp (IN)
Exponent to which the base is raised.

result (OUT)
Output of exponentiation.

Returns

This function returns an error if either of the NUMBER arguments is NULL.

Related Functions

OCIErrorGet(), OCINumberPower()
OCINumberIsInt()

Purpose
Tests if an OCINumber is an integer.

Syntax

```c
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 Functions

- OCIErrorGet(), OCINumberRound(), OCINumberTrunc()
OCINumberIsZero()

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 Functions
OCIErrorGet(), OCINumberSetZero()
OCINumberLn()

Purpose

Takes the natural logarithm (base e) of an Oracle NUMBER.

Syntax

```c
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 Functions

OCIErrorGet(), OCINumberExp(), OCINumberLog()
OCI NUMBER Functions

OCINumberLog()

Purpose

Takes the logarithm, to any base, of an Oracle NUMBER.

Syntax

```c
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 Functions

- `OCIErrorGet()`, `OCINumberLn()`
**OCINumberMod()**

**Purpose**

Gets the modulus (remainder) of the division of two Oracle NUMBERs.

**Syntax**

```
sword OCINumberMod ( OCIError *err,
                    const OCINumber *number1,
                    const OCINumber *number2,
                    OCINumber *result );
```

**Parameters**

- **err (IN/OUT)**
  The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

- **number1 (IN)**
  Pointer to the numerator.

- **number2 (IN)**
  Pointer to the denominator.

- **result (OUT)**
  Remainder of the result.

**Returns**

This function returns an error if `number1` or `number2` is `NULL`, or if there is a divide-by-zero error.

**Related Functions**

- `OCIErrorGet()`, `OCINumberDiv()`
OCINumberMul()

Purpose

Multiplies two Oracle NUMBERS.

Syntax

sword OCINumberMul ( OCIError *err,
const OCINumber *number1,
const OCINumber *number2,
OCINumber *result );

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number1 (IN)
NUMBER to multiply.

number2 (IN)
NUMBER to multiply.

result (OUT)
Multiplication result.

Comments

Multiplies number1 with number2 and returns the result in result.

Returns

This function returns an error if any of the NUMBER arguments is NULL.

Related Functions

OCIErrorGet(), OCINumberDiv()
OCINumberNeg()

Purpose

Negates an Oracle NUMBER.

Syntax

```c
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 Functions

- `OCIErrorGet()`, `OCINumberAbs()`, `OCINumberSign()`
OCI NUMBER Functions

OCINumberPower()

Purpose

Raises a given base to a given exponent.

Syntax

sword OCINumberPower ( OCIError              *err,
                        const OCINumber       *base,
                        const OCINumber       *number,
                        OCINumber             *result );

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

base (IN)
Base of the exponentiation.

number (IN)
Exponent to which the base is to be raised.

result (OUT)
Output of exponentiation.

Returns

This function returns an error if any of the NUMBER arguments is NULL.

Related Functions

OCIErrorGet(), OCINumberExp()
**OCINumberPrec()**

**Purpose**

Rounds an OCINumber to a specified number of decimal digits.

**Syntax**

```c
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 Functions**

- `OCIErrorGet()`
- `OCINumberRound()`
OCI NUMBER Functions

OCINumberRound()

Purpose

Rounds an Oracle NUMBER to a specified decimal place.

Syntax

sword OCINumberRound ( OCIError              *err,
                         const OCINumber       *number,
                         sword                 decplace,
                         OCINumber             *result );

Parameters

er (IN/OUT)
The OCI 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 Functions

OCIErrorGet(), OCINumberTrunc()
OCINumberSetPi()

**Purpose**

Sets an **OCINumber** to pi.

**Syntax**

```c
void OCINumberSetPi ( OCIError *err,
                     OCINumber *num );
```

**Parameters**

- `err (IN/OUT)` - The OCI error handle. If there is an error, it is recorded in `err`, and this function returns **OCI_ERROR**. Obtain diagnostic information by calling **OCIErrorGet()**.

- `num (OUT)` - **NUMBER** set to the value of pi.

**Comments**

Initializes the given **NUMBER** to the value of pi.

**Related Functions**

**OCIErrorGet()**
OCINumberSetZero()

Purpose

Initializes an Oracle NUMBER to zero.

Syntax

```c
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 Functions

OCIErrorGet()
**OCINumberShift()**

**Purpose**

Multiplies a `NUMBER` by a power of 10 by shifting it a specified number of decimal places.

**Syntax**

```c
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 Functions**

- `OCIErrorGet()`
OCINumberSign()

Purpose

Gets sign of an Oracle NUMBER.

Syntax

sword OCINumberSign ( OCIError             *err,
                    const OCINumber      *number,
                    sword                *result );

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Oracle NUMBER whose sign is returned.

result (OUT)
Table 19–13 lists the possible return values.

<table>
<thead>
<tr>
<th>Value of number</th>
<th>Output in result Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>number &lt; 0</td>
<td>-1</td>
</tr>
<tr>
<td>number == 0</td>
<td>0</td>
</tr>
<tr>
<td>number &gt; 0</td>
<td>1</td>
</tr>
</tbody>
</table>

Returns

This function returns an error if number or result is NULL.

Related Functions

OCIErrorGet(), OCINumberAbs()
OCINumberSin()

Purpose
Computes the sine in radians of an Oracle NUMBER.

Syntax
sword OCINumberSin ( OCIError *err,
            const OCINumber *number,
            OCINumber *result );

Parameters
err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the sine in radians.

result (OUT)
Result of the sine.

Returns
This function returns an error if either of the number arguments is NULL.

Related Functions
OCIErrorGet(), OCINumberArcSin()
**OCINumberSqrt()**

**Purpose**
Computes the square root of an Oracle NUMBER.

**Syntax**
```c
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 Functions**
OCIErrorGet(), OCINumberPower()
OCINumberSub()

Purpose

Subtracts two Oracle NUMBERs.

Syntax

```c
sword OCINumberSub ( OCIErrror *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 Functions

OCIErrorGet(), OCINumberAdd()
**OCI NUMBER Functions**

**OCI NumberTan()**

**Purpose**
Computes the tangent in radians of an Oracle NUMBER.

**Syntax**

```c
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 Functions**

OCIErrorGet(), OCINumberArcTan(), OCINumberArcTan2()
**OCINumberToInt()**

**Purpose**

Converts an Oracle NUMBER type to integer.

**Syntax**

```c
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 $x\text{bn}$, 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 Functions**

`OCIErrorGet()`, `OCINumberFromInt()`
OCINumberToReal()

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 NUMBERS 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 Functions

`OCIErrorGet()`, `OCINumberFromReal()`
OCINumberToRealArray()

Purpose

Converts an array of NUMBER to an array of real type.

Syntax

```c
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 Functions

`OCIErrorGet()`, `OCINumberToReal()`
OCINumberToText()

Purpose

Converts an Oracle NUMBER to a character string according to a specified format.

Syntax

sword OCINumberToText ( OCIError *err,
const OCINumber *number,
const OraText *fmt,
ub4 fmt_length,
const OraText *nls_params,
ub4 nls_p_length,
ub4 *buf_size,
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
OCINumberToText()

- An invalid format or invalid multibyte format is passed
- A number to text translation for given format causes an overflow

**Related Functions**

OCIErrorGet(), OCINumberFromText()
OCINumberTrunc()

Purpose

Truncates an Oracle NUMBER at a specified decimal place.

Syntax

sword OCINumberTrunc ( OCIError *err,
                        const OCINumber *number,
                        sword decplace,
                        OCINumber *result );

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Input NUMBER.

decplace (IN)
Number of decimal digits to the right of the decimal point at which to truncate. Negative values are allowed.

result (OUT)
Output of truncation.

Returns

This function returns an error if any of the NUMBER arguments is NULL.

Related Functions

OCIErrorGet(), OCINumberRound()
OCI Raw Functions

Table 19–14 describes the OCI raw functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIRawAllocSize()&quot; on page 19-135</td>
<td>Get allocated size of raw memory in bytes</td>
</tr>
<tr>
<td>&quot;OCIRawAssignBytes()&quot; on page 19-136</td>
<td>Assign raw bytes to raw</td>
</tr>
<tr>
<td>&quot;OCIRawAssignRaw()&quot; on page 19-137</td>
<td>Assign raw to raw</td>
</tr>
<tr>
<td>&quot;OCIRawPtr()&quot; on page 19-138</td>
<td>Get raw data pointer</td>
</tr>
<tr>
<td>&quot;OCIRawResize()&quot; on page 19-139</td>
<td>Resize memory of variable-length raw</td>
</tr>
<tr>
<td>&quot;OCIRawSize()&quot; on page 19-140</td>
<td>Get raw size</td>
</tr>
</tbody>
</table>
OCIRawAllocSize()

Purpose

Gets the allocated size of raw memory in bytes.

Syntax

```c
sword OCIRawAllocSize ( OCIEnv         *env,
                          OCIError      *err,
                          const OCIRaw  *raw,
                          ub4           *allocsize );
```

Parameters

- **env (IN/OUT)**
The OCI environment handle initialized in object mode.
- **err (IN/OUT)**
The OCI 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 Functions

- `OCIErrorGet()`, `OCIRawResize()`, `OCIRawSize()`
OCIRawAssignBytes()

Purpose

Assigns raw bytes of type ub1* to Oracle OCIRaw* data type.

Syntax

```c
sword OCIRawAssignBytes ( OCIEnv *env, OCIError *err, const ub1 *rhs, ub4 rhs_len, OCIRaw *lhs );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  See Also: "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 Functions

OCIErrorGet(), OCIRawAssignRaw()
OCI Raw Functions

OCI Raw Assign Raw

Purpose
Assigns one Oracle RAW data type to another Oracle RAW data type.

Syntax

```c
sword OCIRawAssignRaw ( OCIEnv *env, OCIError *err, const OCIRaw *rhs, OCIRaw **lhs );
```

Parameters

`env (IN/OUT)`
The OCI environment handle initialized in object mode.

See Also: "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

`err (IN/OUT)`
The OCI 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 Functions
OCIErrorGet(), OCIRawAssignBytes()
OCIRawPtr()

Purpose

Gets the pointer to raw data.

Syntax

```c
ub1 *OCIRawPtr ( OCIEnv             *env,
                   const OCIRaw       *raw );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  See Also: "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()"
  on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **raw (IN)**
  Pointer to the data of a given raw.

Comments

None.

Related Functions

OCIErrorGet(), OCIRawAssignRaw()
OCIRawResize()

Purpose

Resizes the memory of a given variable-length raw.

Syntax

```c
sword OCIRawResize ( OCIEnv *env, OCIError *err, ub2 new_size, OCIRaw **raw );
```

Parameters

- **env (IN/OUT)**
The OCI environment handle initialized in object mode.

  `See Also:` "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
The OCI 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 Functions

- `OCIErrorGet()`, `OCIRawAllocSize()`, `OCIRawSize()`
OCIRawSize()

Purpose

Returns the size of a given raw in bytes.

Syntax

```
ub4 OCIRawSize ( OCIEnv *env,
                 const OCIRaw *raw );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  See Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **raw (IN/OUT)**
  Raw whose size is returned.

Comments

None.

Related Functions

OCIErrorGet(), OCIRawAllocSize(), OCIRawResize()
Table 19–15 describes the OCI Reference (REF) functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIRefAssign()&quot; on page 19-142</td>
<td>Assign one REF to another</td>
</tr>
<tr>
<td>&quot;OCIRefClear()&quot; on page 19-143</td>
<td>Clear or nullify a REF</td>
</tr>
<tr>
<td>&quot;OCIRefFromHex()&quot; on page 19-144</td>
<td>Convert hexadecimal string to REF</td>
</tr>
<tr>
<td>&quot;OCIRefHexSize()&quot; on page 19-145</td>
<td>Return size of hexadecimal representation of REF</td>
</tr>
<tr>
<td>&quot;OCIRefIsEqual()&quot; on page 19-146</td>
<td>Compare two REFs for equality</td>
</tr>
<tr>
<td>&quot;OCIRefIsNull()&quot; on page 19-147</td>
<td>Test if a REF is NULL</td>
</tr>
<tr>
<td>&quot;OCIRefToHex()&quot; on page 19-148</td>
<td>Convert REF to hexadecimal string</td>
</tr>
</tbody>
</table>
OCIRefAssign()

Purpose

Assigns one REF to another, such that both reference the same object.

Syntax

```c
sword OCIRefAssign ( OCIEnv *env,
                     OCIError *err,
                     const OCIRef *source,
                     OCIRef **target );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

- **err (IN/OUT)**
  The OCI 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 Functions

`OCIErrorGet()`, `OCIRefIsEqual()`
OCIRefClear()

Purpose

Clears or NULLifies a given REF.

Syntax

void OCIRefClear ( OCIEnv        *env,
                   OCIRef        *ref );

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

See Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

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 Functions

OCIErrorGet(), OCIRefIsNull()
OCIRefFromHex()

Purpose

Converts the given hexadecimal string into a REF.

Syntax

```c
sword OCIRefFromHex ( OCIEnv              *env,
                    OCIError            *err,
                    const OCISvcCtx     *svc,
                    const OraText       *hex,
                    ub4                 length,
                    OCIRef              **ref );
```

Parameters

- **env (IN/OUT)**
The OCI environment handle initialized in object mode.

  See Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()"
on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
The OCI 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 Functions

OCIErrorGet(), OCIRefToHex()
OCIRefHexSize()

Purpose

Returns the size of the hexadecimal representation of a REF.

Syntax

```c
ub4 OCIRefHexSize ( OCIEnv *env,
                    const OCIRef *ref );
```

Parameters

- **env (IN/OUT)**
The OCI environment handle initialized in object mode.

  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **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 Functions

OCIErrorGet(), OCIRefFromHex()
OCIRefIsEqual()

Purpose

Compares two REFs to determine if they are equal.

Syntax

```c
boolean OCIRefIsEqual ( OCIEnv *env, 
                        const OCIRef *x, 
                        const OCIRef *y );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  See Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()"
  on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **x (IN)**
  REF to compare.

- **y (IN)**
  REF to compare.

Comments

Two REFs are equal if and only if they are both referencing the same object, whether persistent or transient.

**Note:** Two NULL REFs are considered not equal by this function.

Returns

TRUE, if the two REFs are equal.

FALSE, if the two REFs are not equal, or x is NULL, or y is NULL.

Related Functions

OCIErrorGet(), OCIRefAssign()
OCIRefIsNull()

Purpose

Tests if a REF is NULL.

Syntax

```c
boolean OCIRefIsNull ( OCIEnv *env,
                       const OCIRef *ref );
```

Parameters

- **env (IN/OUT)**
The OCI environment handle initialized in object mode.

  See Also: "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **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 Functions

OCIErrorGet(), OCIRefClear()
OCIRefToHex()

Purpose

Converts a REF to a hexadecimal string.

Syntax

```c
sword OCIRefToHex ( OCIEnv *env, OCIError *err, const OCIRef *ref, OraText *hex, ub4 *hex_length );
```

Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode.

**See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

**err (IN/OUT)**
The OCI 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 Functions

OCIErrorGet(), OCIRefFromHex(), OCIRefHexSize(), OCIRefIsNull()
OCI String Functions

Table 19–16 describes the OCI string functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIStringAllocSize()&quot; on page 19-150</td>
<td>Get the allocated size of string memory in bytes</td>
</tr>
<tr>
<td>&quot;OCIStringAssign()&quot; on page 19-151</td>
<td>Assign a string to a string</td>
</tr>
<tr>
<td>&quot;OCIStringAssignText()&quot; on page 19-152</td>
<td>Assign a text string to a string</td>
</tr>
<tr>
<td>&quot;OCIStringPtr()&quot; on page 19-153</td>
<td>Get a string pointer</td>
</tr>
<tr>
<td>&quot;OCIStringResize()&quot; on page 19-154</td>
<td>Resize the string memory</td>
</tr>
<tr>
<td>&quot;OCIStringSize()&quot; on page 19-155</td>
<td>Get the string size</td>
</tr>
</tbody>
</table>
OCIStringAllocSize()

Purpose

Gets the allocated size of string memory in code points (Unicode) or in bytes.

Syntax

```c
sword OCIStringAllocSize ( OCIEnv              *env,
                             OCIError            *err,
                             const OCIString     *vs,
                             ub4                 *allocsize );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 Functions

`OCIErrorGet()`, `OCIStringResize()`, `OCIStringSize()`
OCI String Functions

OCIStringAssign()

Purpose

Assigns one string to another string.

Syntax

```c
sword OCIStringAssign ( OCIEnv               *env,
                         OCIError             *err,
                         const OCIString      *rhs,
                         OCIString            **lhs );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  See Also:  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()"
  on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 Functions

- `OCIErrorGet()`, `OCIStringAssignText()`
OCIStringAssignText()

Purpose

Assigns the source text string to the target string.

Syntax

```c
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 Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 Functions

OCIErrorGet(), OCIStringAssign()
OCIStringPtr()

Purpose

Gets a pointer to the text of a given string.

Syntax

```c
text *OCIStringPtr ( OCIEnv *env, const OCIString *vs );
```

Parameters

**env (IN/OUT)**
The OCI environment handle initialized in object mode.

**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 Functions

- OCIErrorGet()
- OCIStringAssign()
OCIStringResize()

Purpose

Resizes the memory of a given string.

Syntax

```c
sword OCIStringResize ( OCIEnv *env,
  OCIError *err,
  ub4 new_size,
  OCIString **str );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.
  
  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()"
  on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 Functions

```
OCIErrorGet(), OCIStringAllocSize(), OCIStringLength()
```
OCIStringSize()  

Purpose  

Gets the size of the given string vs.

Syntax  

```c
ub4 OCIStringSize ( OCIEnv *env,  
                   const OCIString *vs );
```

Parameters  

**env (IN/OUT)**  
The OCI environment handle initialized in object mode.

**See Also:**  "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

**vs (IN)**  
String whose size is returned, in number of bytes.

Comments  

The returned size does not include an extra byte for NULL termination.

Related Functions  

OCIErrorGet(), OCIStringResize()
OCI Table Functions

Table 19–17 describes the OCI table functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCITableDelete()&quot; on page 19-157</td>
<td>Delete element</td>
</tr>
<tr>
<td>&quot;OCITableExists()&quot; on page 19-158</td>
<td>Test whether element exists</td>
</tr>
<tr>
<td>&quot;OCITableFirst()&quot; on page 19-159</td>
<td>Return first index of table</td>
</tr>
<tr>
<td>&quot;OCITableLast()&quot; on page 19-160</td>
<td>Return last index of table</td>
</tr>
<tr>
<td>&quot;OCITableNext()&quot; on page 19-161</td>
<td>Return next available index of table</td>
</tr>
<tr>
<td>&quot;OCITablePrev()&quot; on page 19-162</td>
<td>Return previous available index of table</td>
</tr>
<tr>
<td>&quot;OCITableSize()&quot; on page 19-163</td>
<td>Return current size of table</td>
</tr>
</tbody>
</table>
OCI Table Functions

OCITableDelete()

Purpose

Deletes the element at the specified index.

Syntax

```c
sword OCITableDelete ( OCIEnv          *env,
                      OCIError        *err,
                      sb4             index,
                      OCITable        *tbl );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 Functions

- `OCIErrorGet()`, `OCITableExists()`
OCITableExists()

Purpose

Tests whether an element exists at the given index.

Syntax

```c
sword OCITableExists ( OCIEnv      *env,
                         OCIError    *err,
                         const OCITable *tbl,
                         sb4         index,
                         boolean     *exists);
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 Functions

OCIErrorGet(), OCITableDelete()
**OCI Table Functions**

### OCITableFirst()

**Purpose**

Returns the index of the first existing element in a given table.

**Syntax**

```c
sword OCITableFirst ( OCIEnv *env, OCIError *err, const OCITable *tbl, sb4 *index );
```

**Parameters**

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 Functions**

`OCIErrorGet()`, `OCITableDelete()`, `OCITableLast()`
OCITableLast()

Purpose

Returns the index of the last existing element of a table.

Syntax

```c
sword OCITableLast ( OCIEnv *env, OCIError *err, const OCITable *tbl, sb4 *index );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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.

Returns

This function returns an error if the table is empty.

Related Functions

`OCIErrorGet()`, `OCITableFirst()`, `OCITableNext()`, `OCITablePrev()`
OCI Table Functions

OCI Table Functions

OCI Table Next()

Purpose

Returns the index of the next existing element of a table.

Syntax

sword OCITableNext ( OCIEnv *env, 
          OCIError *err, 
          sb4 index, 
          const OCITable *tbl, 
          sb4 *next_index 
          boolean *exists );

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

See Also:    "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

er (IN/OUT)
The OCI 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.

See Also: The description of OCITableSize() for information about the existence of non-data holes (deleted elements) in tables

Related Functions

OCIErrorGet(), OCITablePrev()
OCITablePrev()

Purpose

Returns the index of the previous existing element of a table.

Syntax

```c
sword OCITablePrev ( OCIEnv *env, OCIError *err, sb4 index, const OCITable *tbl, sb4 *prev_index, boolean *exists );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

  **See Also:** "OCIEnvCreate()" on page 16-13, "OCIEnvNlsCreate()" on page 16-17, and "OCIInitialize()" on page E-5 (deprecated)

- **err (IN/OUT)**
  The OCI 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 \( \text{index} \), such that \( \text{exists}(j) \) is TRUE.

**See Also:** The description of OCITableSize() for information about the existence of non-data holes (deleted elements) in tables

Related Functions

OCIErrorGet(), OCITableNext()
OCITableSize()

Purpose

Returns the size of the given table, not including any holes created by deleted elements.

Syntax

```c
sword OCITableSize ( OCIEnv              *env,
                     OCIError           *err,
                     const OCITable     *tbl
                      sb4                *size );
```

Parameters

- **env (IN/OUT)**
  The OCI environment handle initialized in object mode.

- **err (IN/OUT)**
  The OCI 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()`.

Example 19–4 shows a code fragment where an element is deleted from a nested table.

**Example 19–4  Deleting an Element from a Nested table**

```c
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 Example 19–5. Continuing Example 19–4.

**Example 19–5  Getting a Count of All Elements Including Deleted Elements from a Nested Table**

```c
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 Functions

OCIErrorGet(), OCICollSize()
This chapter presents the cartridge functions. For code examples, see the demonstration programs included with your Oracle Database installation. For additional information, see Appendix B.

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

Introduction to External Procedure and Cartridge Services Functions

This chapter first describes the OCI external procedure functions. These functions enable users of external procedures to raise errors, allocate some memory, and get OCI context information. For more information about using these functions in external procedures, see the Oracle Database Advanced Application Developer’s Guide.

Then the cartridge services functions are described. For more information about using these functions, see Oracle Database Data Cartridge Developer’s Guide.

Conventions for OCI Functions

See the "Conventions for OCI Functions" on page 16-1 for the conventions used in describing each function. The entries for each function may also contain the following information:

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:

```c
OCIExtProcContext *with_context;
```
Table 20–1 lists the OCI external procedure functions for C that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIExtProcAllocCallMemory()&quot; on page 20-4</td>
<td>Allocate memory for the duration of the External Procedure</td>
</tr>
<tr>
<td>&quot;OCIExtProcGetEnv()&quot; on page 20-5</td>
<td>Get the OCI environment, service context, and error handles</td>
</tr>
<tr>
<td>&quot;OCIExtProcRaiseExcp()&quot; on page 20-6</td>
<td>Raise an Exception to PL/SQL</td>
</tr>
<tr>
<td>&quot;OCIExtProcRaiseExcpWithMsg()&quot; on page 20-7</td>
<td>Raise an exception with a message</td>
</tr>
</tbody>
</table>
OCIExtProcAllocCallMemory()

Purpose
Allocate a block of memory of size parameter IN and return an untyped (opaque) pointer to the block. The block is freed when the external procedure returns.

Syntax

```c
void * OCIExtProcAllocCallMemory ( OCIExtProcContext *with_context,
                                  size_t amount );
```

Parameters

- `with_context`: The pointer to the `with_context` parameter that is passed to the external procedure.
- `amount`: 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

```c
Example 20–1  Using OCIExtProcAllocCallMemory() to Allocate 1024 Bytes of Memory

text *ptr = (text *)OCIExtProcAllocCallMemory(wctx, 1024)
```

Related Functions

- `OCIErrorGet()`, `OCIAlloc()`
OCIExtProcGetEnv()

Purpose

Gets the OCI environment, service context, and error handles.

Syntax

```c
sword OCIExtProcGetEnv ( OCIExtProcContext    *with_context,
                        OCIEnv               **envh,
                        OCISvcCtx            **svch,
                        OCIError             **errh );
```

Parameters

- `with_context (IN)`
  The `with_context` pointer that is passed to the C external procedure. See "With_Context Type" on page 20-2.

- `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:

```c
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 Functions

`OCIEnvCreate()`, `OCIArrayDescriptorAlloc()`, `OCIHandleAlloc()`
**OCIExtProcRaiseExcp()**

**Purpose**

Raises an Exception to PL/SQL.

**Syntax**

```c
size_t OCIExtProcRaiseExcp ( OCIExtProcContext *with_context, int errnum );
```

**Parameters**

- **with_context (IN)**
  The `with_context` pointer that is passed to the C external procedure.

  See Also:  "With_Context Type" on page 20-2

- **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 Functions**

OCIExtProcRaiseExcpWithMsg()
OCIExtProcRaiseExcpWithMsg()

Purpose

 Raises an exception with a message.

Syntax

```c
size_t OCIExtProcRaiseExcpWithMsg ( OCIExtProcContext *with_context,
                                     int errnum,
                                     char *errmsg,
                                     size_t msglen );
```

Parameters

- **with_context (IN)**
  The `with_context` pointer that is passed to the C external procedure.

  **See Also:** "With_Context Type" on page 20-2

- **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.

  **See Also:** The description of OCIExtProcRaiseExcp()

Returns

This function returns `OCIEXTPROC_SUCCESS` if the call succeeds. It returns `OCIEXTPROC_ERROR` if the call fails.

Related Functions

- OCIExtProcRaiseExcp()
Cartridge Services — Memory Services

Table 20–2 lists the memory services functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIDurationBegin()&quot; on page 20-9</td>
<td>Start a user duration</td>
</tr>
<tr>
<td>&quot;OCIDurationEnd()&quot; on page 20-10</td>
<td>Terminate a user duration</td>
</tr>
<tr>
<td>&quot;OCIMemoryAlloc()&quot; on page 20-11</td>
<td>Allocate memory of a given size from a given duration</td>
</tr>
<tr>
<td>&quot;OCIMemoryFree()&quot; on page 20-12</td>
<td>Free a memory chunk</td>
</tr>
<tr>
<td>&quot;OCIMemoryResize()&quot; on page 20-13</td>
<td>Resize a memory chunk</td>
</tr>
</tbody>
</table>

See Also: Oracle Database Data Cartridge Developer’s Guide for more information about using these functions
OCI(DurationBegin()

**Purpose**

Starts a user duration.

**Syntax**

```c
sword OCI(DurationBegin ( 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 Functions**

OCI(DurationEnd())
OCI Duration End

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.

Related Functions
OCI Duration Begin
OCIMemoryAlloc()

Purpose

Allocates memory of a given size from a given duration.

Syntax

```c
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.
OCIMemoryFree()

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.
OCIMemoryResize()

Purpose
Resizes a memory chunk to a new size.

Syntax
```c
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 theOCI_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.
Table 20–3 lists the maintaining context functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIContextClearValue()&quot; on page 20-15</td>
<td>Remove the value stored in the context</td>
</tr>
<tr>
<td>&quot;OCIContextGenerateKey()&quot; on page 20-16</td>
<td>Return a unique 4-byte value each time it is called</td>
</tr>
<tr>
<td>&quot;OCIContextGetValue()&quot; on page 20-17</td>
<td>Return the value stored in the context</td>
</tr>
<tr>
<td>&quot;OCIContextSetValue()&quot; on page 20-18</td>
<td>Save a value (or address) for a particular duration</td>
</tr>
</tbody>
</table>

See Also: Oracle Database Data Cartridge Developer’s Guide for more information about using these functions
OCIContextClearValue()

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 is 64 bits.

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.
OCIContextGenerateKey()

Purpose

Returns a unique, 4-byte value each time it is called.

Syntax

```c
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`. 
OCIContextGetValue()

Purpose

Returns the value that is stored in the context associated with the given key (by calling OCIContextSetValue()).

Syntax

```c
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 is 64 bits.

**ctx_value (IN)**
Pointer to the value stored in the context (NULL if no value was stored).

Comments

For **ctx_value**, a pointer to a preallocated pointer for the stored context to be returned is required.

Returns

- If the operation succeeds, the function returns **OCI_SUCCESS**.
- If the operation fails, the function returns **OCI_ERROR**.
OCIContextSetValue()

**Purpose**

Saves a value (or address) for a particular duration.

**Syntax**

```c
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 is 64 bits.

- **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**.
Table 20–4 lists the parameter manager interface functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIExtractFromFile()&quot; on page 20-20</td>
<td>Process the keys and their values in the given file</td>
</tr>
<tr>
<td>&quot;OCIExtractFromList()&quot; on page 20-21</td>
<td>Generate a list of values for the parameter denoted by index in the parameter list</td>
</tr>
<tr>
<td>&quot;OCIExtractFromStr()&quot; on page 20-22</td>
<td>Process the keys and their values in the given string</td>
</tr>
<tr>
<td>&quot;OCIExtractInit()&quot; on page 20-23</td>
<td>Initialize the parameter manager</td>
</tr>
<tr>
<td>&quot;OCIExtractReset()&quot; on page 20-24</td>
<td>Reinitialize memory</td>
</tr>
<tr>
<td>&quot;OCIExtractSetKey()&quot; on page 20-25</td>
<td>Register information about a key with the parameter manager</td>
</tr>
<tr>
<td>&quot;OCIExtractSetNumKeys()&quot; on page 20-27</td>
<td>Inform the parameter manager of the number of keys that are to be registered</td>
</tr>
<tr>
<td>&quot;OCIExtractTerm()&quot; on page 20-28</td>
<td>Release all dynamically allocated storage</td>
</tr>
<tr>
<td>&quot;OCIExtractToBool()&quot; on page 20-29</td>
<td>Get the Boolean value for the specified key</td>
</tr>
<tr>
<td>&quot;OCIExtractToInt()&quot; on page 20-30</td>
<td>Get the integer value for the specified key</td>
</tr>
<tr>
<td>&quot;OCIExtractToList()&quot; on page 20-31</td>
<td>Generate a list of parameters from the parameter structures that are stored in memory</td>
</tr>
<tr>
<td>&quot;OCIExtractToOCINum()&quot; on page 20-32</td>
<td>Get the number value for the specified key</td>
</tr>
<tr>
<td>&quot;OCIExtractToStr()&quot; on page 20-33</td>
<td>Get the string value for the specified key</td>
</tr>
</tbody>
</table>

See Also: Oracle Database Data Cartridge Developer’s Guide for more information about using these functions
OCIExtractFromFile()

Purpose

Processes the keys and their values in the given file.

Syntax

```c
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.
OCIExtractFromList()

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.
OCIExtractFromStr()

Purpose

Processes the keys and their values in the given string.

Syntax

```c
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`. 
**OCIExtractInit()**

**Purpose**

Initializes the parameter manager.

**Syntax**

```c
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`
- `OCI_ERROR`
OCIExtractReset()

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.

er (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,


OCIExtractSetKey()

**Purpose**

Registers information about a key with the parameter manager.

**Syntax**

```c
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.
OCIExtractSetKey()

Comments
This routine must be called after calling OCIExtractSetNumKeys() and before calling OCIExtractFromFile() or OCIExtractFromStr().

Returns
OCI_SUCCESS; OCI_INVALID_HANDLE; or OCI_ERROR.
OCIExtractSetNumKeys()

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.
OCIExtractTerm()

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.
OCIExtractToBool()

Purpose

Gets the Boolean value for the specified key. 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.
OCIExtractToInt()

Purpose

Gets the integer value for the specified key. The valno’th value (starting with 0) is returned.

Syntax

```c
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.
OCIExtractToList()  

Purpose  
Generates a list of parameters from the parameter structures that are stored in memory. Must be called before OCIExtractValues() is called.

Syntax  
sword OCIExtractToList( 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.
OCIExtractToOCINum()

Purpose

Gets the OCINumber value for the specified key. The \texttt{valno}'th value (starting with 0) is returned.

Syntax

\begin{verbatim}
sword OCIExtractToOCINum( void *hndl,
OCIError *err,
OraText *keyname,
ulong valno,
OCINumber *retval );
\end{verbatim}

Parameters

\textbf{hndl (IN)}

The OCI environment or user session handle.

\textbf{err (IN/OUT)}

The OCI error handle. If there is an error, it is recorded in \texttt{err}, and this function returns \texttt{OCI_ERROR}. Diagnostic information can be obtained by calling \texttt{OCIErrorGet()}.

\textbf{keyname (IN)}

Key name.

\textbf{valno (IN)}

Which value to get for this key.

\textbf{retval (OUT)}

The actual OCINumber value.

Returns

\texttt{OCI_SUCCESS}; \texttt{OCI_INVALID_HANDLE}; \texttt{OCI_NO_DATA}; or \texttt{OCI_ERROR}.

\texttt{OCI_NO_DATA} means that there is no \texttt{valno}'th value for this key.
OCIExtractToStr()

Purpose

Gets the string value for the specified key. The valno\textsuperscript{th} value (starting with 0) is returned.

Syntax

\begin{verbatim}
sword OCIExtractToStr( void *hndl, OCIError *err, OraText *keyname, uword valno, OraText *retval, uword buflen );
\end{verbatim}

Parameters

\begin{description}
\item[hndl (IN)] The OCI environment or user session handle.
\item[err (IN/OUT)] The OCI error handle. If there is an error, it is recorded in \texttt{err}, and this function returns \texttt{OCI_ERROR}. Diagnostic information can be obtained by calling \texttt{OCIErrorGet()}.
\item[keyname (IN)] Key name.
\item[valno (IN)] Which value to get for this key.
\item[retval (OUT)] The actual \texttt{NULL}-terminated string value.
\item[ buflen ] The length of the buffer for \texttt{retval}.
\end{description}

Returns

\begin{verbatim}
OCI_SUCCESS; OCI_INVALID_HANDLE; OCI_NO_DATA; or OCI_ERROR.
\end{verbatim}

OCI\_NO\_DATA means that there is no valno\textsuperscript{th} value for this key.
Table 20–5 lists the file I/O interface functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIFileClose()&quot; on page 20-35</td>
<td>Close a previously opened file</td>
</tr>
<tr>
<td>&quot;OCIFileExists()&quot; on page 20-36</td>
<td>Test to see if the file exists</td>
</tr>
<tr>
<td>&quot;OCIFileFlush()&quot; on page 20-37</td>
<td>Write buffered data to a file</td>
</tr>
<tr>
<td>&quot;OCIFileGetLength()&quot; on page 20-38</td>
<td>Get the length of a file</td>
</tr>
<tr>
<td>&quot;OCIFileInit()&quot; on page 20-39</td>
<td>Initialize the OCIFile package</td>
</tr>
<tr>
<td>&quot;OCIFileOpen()&quot; on page 20-40</td>
<td>Open a file</td>
</tr>
<tr>
<td>&quot;OCIFileRead()&quot; on page 20-42</td>
<td>Read from a file into a buffer</td>
</tr>
<tr>
<td>&quot;OCIFileSeek()&quot; on page 20-43</td>
<td>Change the current position in a file</td>
</tr>
<tr>
<td>&quot;OCIFileTerm()&quot; on page 20-44</td>
<td>Terminate the OCIFile package</td>
</tr>
<tr>
<td>&quot;OCIFileWrite()&quot; on page 20-45</td>
<td>Write buflen bytes into the file</td>
</tr>
</tbody>
</table>

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().
OCIFileClose()

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.

er (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.
OCIFileExists()

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.
OCIFileFlush()

Purpose

Writes buffered data to a file.

Syntax

```c
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`
- `OCI_ERROR`
OCIFileGetLength()

Purpose

 Gets the length of a file.

Syntax

```c
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`. 
OCIFileInit()

Purpose

Initializes the OCIFile package. It must be called before any other OCIFile routine is called.

Syntax

```c
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`
- `OCI_ERROR`
OCIFileOpen()

Purpose

Opens a file.

Syntax

```c
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.
OCIFileRead()

Purpose

Reads from a file into a buffer.

Syntax

```c
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`. 
OCIFileSeek()

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.
OCIFileTerm()

Purpose
Terminates the OCIFile package. It 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.
OCIFileWrite()

Purpose

Writes buflen bytes into the file.

Syntax

```c
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.
Table 20–6 lists the string formatting functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIFormatInit()&quot; on page 20-47</td>
<td>Initialize the OCIFormat package</td>
</tr>
<tr>
<td>&quot;OCIFormatString()&quot; on page 20-48</td>
<td>Write a text string into the supplied text buffer</td>
</tr>
<tr>
<td>&quot;OCIFormatTerm()&quot; on page 20-53</td>
<td>Terminate the OCIFormat package</td>
</tr>
</tbody>
</table>

**See Also:** Oracle Database Data Cartridge Developer’s Guide for more information about using these functions
OCIFormatInit()

Purpose

Initializes the OCIFormat package.

Syntax

```c
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`.
**OCIFormatString()**

**Purpose**

Writes a text string into the supplied text buffer using the argument list submitted to it and in accordance with the format string given.

**Syntax**

```
sword OCIFormatString( 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 20–7 and Table 20–8.

- **... (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 20–7, can be in any order and are all optional.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>Left-justify the output in the field.</td>
</tr>
<tr>
<td>'+'</td>
<td>Always print a sign ('+' or '-') for numeric types.</td>
</tr>
<tr>
<td>' '</td>
<td>If a number's sign is not printed, then print a space in the sign position.</td>
</tr>
<tr>
<td>'0'</td>
<td>Pad numeric output with zeros, not spaces.</td>
</tr>
</tbody>
</table>

- 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 20–8, 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>
<thead>
<tr>
<th>Codes</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>Single-byte character in the compiler character set</td>
</tr>
<tr>
<td>'d'</td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td>'e'</td>
<td>Exponential (scientific) notation of the form [-]&lt;r&gt;[&lt;d&gt;...]&lt;e&gt;[&lt;d&gt;] &lt;d&gt;&lt;d&gt;&lt;d&gt; &lt;d&gt; where &lt;r&gt; is the radix character for the current language and &lt;d&gt; 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).</td>
</tr>
<tr>
<td>'f'</td>
<td>Fixed decimal notation of the form [-]&lt;d&gt;[&lt;d&gt;...]&lt;r&gt;[&lt;d&gt;...]&lt;d&gt;&lt;d&gt; where &lt;r&gt; is the appropriate radix character for the current language and &lt;d&gt; 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.</td>
</tr>
<tr>
<td>'g'</td>
<td>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</td>
</tr>
<tr>
<td>'i'</td>
<td>Identical to 'd'</td>
</tr>
</tbody>
</table>
Example 20–2 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. */

/* 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 *)"%2u/%02u/%02u", 
    day, month, year);
OCIFormatString() function:

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.

---

OCIFormatUb1():

- **Format**: String formatting function.
- **Usage**: Formats a date string in a specific format.
- **Parameters**:
  - `hndl`: The handle to the OCI environment.
  - `err`: The error code.
  - `buffer`: The buffer where the formatted string will be stored.
  - `returnLen`: The length of the return string.
  - `format`: The format string used for formatting.
  - `day`, `month`, `year`: Variables to hold the day, month, and year values.

---

OCIFormatUb1(day),
OCIFormatUb1(month),
OCIFormatUb1(year),
OCIFormatEnd); /* Buffer is *03/10/97*. */

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.
OCIFormatTerm()

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.
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 Appendix B.

This chapter contains these topics:

- Introduction to Any Type and Data Interfaces
- OCI Type Interface Functions
- OCI Any Data Interface Functions
- OCI Any Data Set Interface Functions

### Introduction to Any Type and Data Interfaces

This chapter describes the OCI Any Type and Data functions in detail.

**See Also:** "AnyType, AnyData, and AnyDataSet Interfaces" on page 12-20

### Conventions for OCI Functions

See the "Conventions for OCI Functions" on page 16-1 for the conventions used in describing each function. The entries for each function may also contain the following information:

### Function Return Values

The OCI Any Type and Data functions typically return one of the values described in Table 21–1.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_SUCCESS</td>
<td>The operation succeeded.</td>
</tr>
<tr>
<td>OCI_ERROR</td>
<td>The operation failed. The specific error can be retrieved by calling OCIErrorGet() on the error handle passed to the function.</td>
</tr>
<tr>
<td>OCI_INVALID_HANDLE</td>
<td>The OCI handle passed to the function is invalid.</td>
</tr>
</tbody>
</table>
See Also: "Error Handling in OCI" on page 2-20 for more information about return codes and error handling
## OCI Type Interface Functions

Table 21–2 lists the Type Interface functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCITypeAddAttr()&quot; on page 21-4</td>
<td>Add an attribute to an object type that was constructed earlier with typecode OCI_TYPECODE_OBJECT</td>
</tr>
<tr>
<td>&quot;OCITypeBeginCreate()&quot; on page 21-5</td>
<td>Begin the construction process for a transient type. The type is anonymous (no name).</td>
</tr>
<tr>
<td>&quot;OCITypeEndCreate()&quot; on page 21-6</td>
<td>Finish construction of a type description. Subsequently, only access is allowed.</td>
</tr>
<tr>
<td>&quot;OCITypeSetBuiltin()&quot; on page 21-7</td>
<td>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).</td>
</tr>
<tr>
<td>&quot;OCITypeSetCollection()&quot; on page 21-8</td>
<td>Set collection type information. This call can be made only if the type has been constructed with a collection typecode.</td>
</tr>
</tbody>
</table>
OCITypeAddAttr()

Purpose

Adds an attribute to an object type that was constructed earlier with typecode OCI_TYPECODE_OBJECT.

Syntax

```
sword OCITypeAddAttr ( OCISvcCtx   *svchp,
                     OCIError    *errhp,
                     OCIType     *type,
                     const text  *a_name,
                     ub4         a_length,
                     OCIParam    *attr_info );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

type (IN/OUT)
The type description that is being constructed.

a_name (IN) [optional]
The name of the attribute.

a_length (IN) [optional]
The length of 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.
OCITypeBeginCreate()

Purpose

Begins the construction process for a transient type. The type is anonymous (no name).

Syntax

```c
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" on page 3-25. Additional information about built-in types (precision, scale for numbers, character set information for VARCHAR2s, 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.
OCITypeEndCreate()

Purpose

Finishes construction of a type description. Subsequently, only access is allowed.

Syntax

```c
sword OCITypeEndCreate ( OCISvcCtx  *svchp,
                          OCIError   *errhp,
                          OCIType    *type );
```

Parameters

- **svchp (IN)**
The OCI service context.

- **errhp (IN/OUT)**
The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

- **type (IN/OUT)**
The type description that is being constructed.
OCITypeSetBuiltin()

Purpose
Sets built-in type information. This call can be made only if the type has been constructed with a built-in typecode (OCI_TYPECODE_NUMBER, and so on).

Syntax
sword OCITypeSetBuiltin ( OCISvcCtx *svchp, OCIError *errhp, OCIType *type, OCIParam *builtin_info );

Parameters
svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

type (IN/OUT)
The type description that is being constructed.

builtin_info (IN)
Provides information 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.
OCITypeSetCollection()

Purpose
Sets collection type information. This call can be made only if the type has been constructed with a collection typecode.

Syntax
sword OCITypeSetCollection ( OCISvcCtx   *svchp,
                           OCIError    *errhp,
                           OCIType     *type,
                           OCIParam    *collelem_info,
                           ub4         coll_count );

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

type (IN/OUT)
The type descriptor that is being constructed.

collelem_info (IN)
collelem_info provides information about the collection element. It is obtained by allocating an OCIParam parameter handle and setting type information in the OCIParam using OCIAttrSet() calls.

coll_count (IN)
The count of elements in the collection. Pass 0 for a nested table (which is unbounded).
## OCI Any Data Interface Functions

Table 21–3 lists the Any Data Interface functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>“OCIAnyDataAccess()” on page 21-10</td>
<td>Retrieve the data value of an OCIAnyData</td>
</tr>
<tr>
<td>“OCIAnyDataAttrGet()” on page 21-12</td>
<td>Get the value of the attribute at the current position in the OCIAnyData</td>
</tr>
<tr>
<td>“OCIAnyDataAttrSet()” on page 21-14</td>
<td>Set the attribute at the current position with a given value</td>
</tr>
<tr>
<td>“OCIAnyDataBeginCreate()” on page 21-16</td>
<td>Allocate an OCIAnyData for the given duration and initialize it with the type information</td>
</tr>
<tr>
<td>“OCIAnyDataCollAddElem()” on page 21-18</td>
<td>Add the next collection element to the collection attribute of the OCIAnyData at the current attribute position</td>
</tr>
<tr>
<td>“OCIAnyDataCollGetElem()” on page 21-20</td>
<td>Access sequentially the elements in the collection attribute at the current position in the OCIAnyData</td>
</tr>
<tr>
<td>“OCIAnyDataConvert()” on page 21-22</td>
<td>Construct an OCIAnyData with the given data value of the given type</td>
</tr>
<tr>
<td>“OCIAnyDataDestroy()” on page 21-24</td>
<td>Free an AnyData</td>
</tr>
<tr>
<td>“OCIAnyDataEndCreate()” on page 21-25</td>
<td>Mark the end of OCIAnyData creation</td>
</tr>
<tr>
<td>“OCIAnyDataGetCurrAttrNum()” on page 21-26</td>
<td>Return the current attribute number of the OCIAnyData</td>
</tr>
<tr>
<td>“OCIAnyDataGetType()” on page 21-27</td>
<td>Get the type corresponding to an AnyData value</td>
</tr>
<tr>
<td>“OCIAnyDataIsNull()” on page 21-28</td>
<td>Check if OCIAnyData is NULL</td>
</tr>
<tr>
<td>“OCIAnyDataTypeCodeToSqlt()” on page 21-29</td>
<td>Convert the OCITypeCode for an AnyData value to the SQT code that corresponds to the representation of the value as returned by the OCIAnyData API</td>
</tr>
</tbody>
</table>
OCIAnyDataAccess()

Purpose
Retrieves the data value of an OCIAnyData. 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.
OCI Any Data Attr Get()

Purpose

Gets the value of the attribute at the current position in the OCI Any Data. Attribute values can be accessed sequentially.

Syntax

```c
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 `OCIErrGet()`.

- **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 21–4 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>
<thead>
<tr>
<th>Data Types</th>
<th>attr_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2, VARCHAR, CHAR</td>
<td>OCIString **</td>
</tr>
<tr>
<td>NUMBER, REAL, INT, FLOAT, DECIMAL</td>
<td>OCINumber **</td>
</tr>
<tr>
<td>DATE</td>
<td>OCIDate **</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>OCIDateTime **</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>OCIDateTime **</td>
</tr>
<tr>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>OCIDateTime **</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>OCIInterval **</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>OCIInterval **</td>
</tr>
<tr>
<td>BLOB</td>
<td>OCILOBLocator ** or OCIBlobLocator **</td>
</tr>
<tr>
<td>CLOB</td>
<td>OCILOBLocator ** or OCIClobLocator *</td>
</tr>
<tr>
<td>BFILE</td>
<td>OCILOBLocator **</td>
</tr>
<tr>
<td>REF</td>
<td>OCIRef **</td>
</tr>
<tr>
<td>RAW</td>
<td>OCIRaw **</td>
</tr>
<tr>
<td>VARRAY</td>
<td>OCIArray **(or OCIAnyData * if is_any is TRUE)</td>
</tr>
<tr>
<td>TABLE</td>
<td>OCITable **(or OCIAnyData * if is_any is TRUE)</td>
</tr>
<tr>
<td>OBJECT</td>
<td>void **(or OCIAnyData * if is_any is TRUE)</td>
</tr>
</tbody>
</table>
OCIAnyDataAttrSet()

Purpose

Sets the attribute at the current position with a given value.

Syntax

```c
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 21–5 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>
<thead>
<tr>
<th>Data Types</th>
<th>attr_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2, VARCHAR, CHAR</td>
<td>OCIString *</td>
</tr>
<tr>
<td>NUMBER, REAL, INT, FLOAT, DECIMAL</td>
<td>OCINumber *</td>
</tr>
<tr>
<td>DATE</td>
<td>OCIDate *</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>OCIDateTime *</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>OCIDateTime *</td>
</tr>
<tr>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>OCIDateTime *</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>OCIInterval *</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>OCIInterval *</td>
</tr>
<tr>
<td>BLOB</td>
<td>OCILObLocator * or OCIBlobLocator *</td>
</tr>
<tr>
<td>CLOB</td>
<td>OCILObLocator * or OCIClobLocator *</td>
</tr>
<tr>
<td>BFILE</td>
<td>OCILObLocator *</td>
</tr>
<tr>
<td>REF</td>
<td>OCIRef *</td>
</tr>
<tr>
<td>RAW</td>
<td>OCIRaw *</td>
</tr>
<tr>
<td>VARRAY</td>
<td>OCIArray * (or OCIAnyData * if is_any is TRUE)</td>
</tr>
<tr>
<td>TABLE</td>
<td>OCITable * (or OCIAnyData * if is_any is TRUE)</td>
</tr>
<tr>
<td>OBJECT</td>
<td>void * (or OCIAnyData * if is_any is TRUE)</td>
</tr>
</tbody>
</table>
OCIAnyDataBeginCreate()

Purpose

Allocates an OCIAnyData for the given duration and initializes it with the type information.

Syntax

```c
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 OCIDurationBegin().
- A predefined duration, such as OCI_DURATION_SESSION.

`sdata` (OUT)
Initialized OCIAnyData. If (*sdata) is not NULL at the beginning of the call, the memory could be reused instead of reallocating space for 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).
**OCIAnyDataCollAddElem()**

**Purpose**

Adds the next collection element to the collection attribute of the OCIAnyData at the current attribute position. If the OCIAnyData is of a collection type, then there is no notion of attribute position and this call adds the next collection element.

**Syntax**

```c
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()" on page 21-14 for the type of attribute to be passed in for all the possible types of the collection elements.
OCIAnyDataCollGetElem()

Purpose

Accesses sequentially the elements in the collection attribute at the current position in the OCIAnyData.

Syntax

```c
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()" on page 21-12 for the type of attribute to be passed in for all the possible types of the collection elements.
OCIAnyDataConvert()

Purpose
Constructs an OCIAnyData with the given data value that is of the given type. 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).
OCIAnyDataDestroy()

Purpose

Frees an OCIAnyData.

Syntax

sword OCIAnyDataDestroy ( OCISvcCtx      *svchp,
                        OCIError       *errhp,
                        OCIAnyData     *sdata );

Parameters

svchp (IN)
The OCI service context.

erphp (IN/OUT)
The OCI 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.
OCIAnyDataEndCreate()

Purpose
Marks the end of OCIAnyData creation. It should be called after initializing all attributes of its instances with suitable values. This call is valid only if OCIAnyDataBeginCreate() 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.
OCIAnyDataGetCurrAttrNum()

Purpose

Returns the current attribute number of OCIAnyData. 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.

erhpp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

sdata (IN)
Initialized OCIAnyData.

attrnum (OUT)
The attribute number.
OCIAnyDataGetType()

Purpose

Gets the type corresponding to an OCIAnyData value. 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

```c
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.
OCIAnyDataIsNull()

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.
OCIAnyDataTypeCodeToSqlt()

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

```c
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.

See Also: "NCHAR Typecodes for OCIAnyData Functions" on page 12-24
## OCI Any Data Set Interface Functions

Table 21–6 lists the Any Data Set Interface functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIAnyDataSetAddInstance()&quot; on page 21-31</td>
<td>Add a new skeleton instance to the OCIAnyDataSet and set all the attributes of the instance to NULL</td>
</tr>
<tr>
<td>&quot;OCIAnyDataSetBeginCreate()&quot; on page 21-32</td>
<td>Allocate an OCIAnyDataSet for the given duration and initialize it with the type information</td>
</tr>
<tr>
<td>&quot;OCIAnyDataSetDestroy()&quot; on page 21-33</td>
<td>Free the OCIAnyDataSet</td>
</tr>
<tr>
<td>&quot;OCIAnyDataSetEndCreate()&quot; on page 21-34</td>
<td>Mark the end of OCIAnyDataSet creation</td>
</tr>
<tr>
<td>&quot;OCIAnyDataSetGetCount()&quot; on page 21-35</td>
<td>Get the number of instances in the OCIAnyDataSet</td>
</tr>
<tr>
<td>&quot;OCIAnyDataSetGetInstance()&quot; on page 21-36</td>
<td>Return the OCIAnyData corresponding to an instance at the current position and update the current position</td>
</tr>
<tr>
<td>&quot;OCIAnyDataSetGetType()&quot; on page 21-37</td>
<td>Get the type corresponding to an OCIAnyDataSet</td>
</tr>
</tbody>
</table>
OCIAnyDataSetAddInstance()

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.
OCIAnyDataSetBeginCreate()

Purpose

Allocates an OCIAnyDataSet for the given duration and initializes it with the type information. The OCIAnyDataSet can hold multiple instances of the given type.

Syntax

```c
sword OCIAnyDataSetBeginCreate ( OCISvcCtx      *svchp,
                               OCIError       *errhp,
                               OCITypeCode    typecode,
                               const OCIType  *type,
                               OCIDuration    dur,
                               OCIAnyDataSet  **data_set );
```

Parameters

svchp (IN)
The OCI service context.

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.

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.

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().

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).

21-32 Oracle Call Interface Programmer's Guide
OCIAnyDataSetDestroy()

Purpose

Frees the OCIAnyDataSet.

Syntax

sword OCIAnyDataSetDestroy ( OCISvcCtx      *svchp,
                             OCIError       *errhp,
                             OCIAnyDataSet  *data_set );

Parameters

svchp (IN)
The OCI service context.

erphp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

data_set (IN/OUT)
OCIAnyDataSet to be freed.
OCIAnyDataSetEndCreate()

Purpose

Marks the end of OCIAnyDataSet creation. This function should be called after constructing all of its instances.

Syntax

```c
sword OCIAnyDataSetEndCreate ( OCISvcCtx *svchp,
                                OCIError   *errhp,
                                OCIAnyDataSet *data_set );
```

Parameters

- **svchp (IN)**
  The OCI service context.

- **errhp (IN/OUT)**
  The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

- **data_set (IN/OUT)**
  Initialized OCIAnyDataSet.
OCIAnyDataSetGetCount()

Purpose

Gets the number of instances in the OCIAnyDataSet.

Syntax

```c
sword OCIAnyDataSetGetCount( OCISvcCtx      *svchp,
                              OCIError       *errhp,
                              OCIAnyDataSet  *data_set,
                              ub4            *count );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

data_set (IN/OUT)
A well-formed OCIAnyDataSet.

count (OUT)
Number of instances in OCIAnyDataSet.
OCIAnyDataSetGetInstance()

Purpose

Returns the OCIAnyData corresponding to an instance at the current position and updates the current position.

Syntax

```c
sword OCIAnyDataSetGetInstance ( OCISvcCtx      *svchp,
                            OCIError       *errhp,
                            OCIAnyDataSet  *data_set,
                            OCIAnyData     **data );
```

Parameters

- **svchp (IN)**
The OCI service context.

- **errhp (IN/OUT)**
The OCI error handle. If there is an error, it is recorded in err, and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

- **data_set (IN/OUT)**
A well-formed OCIAnyDataSet.

- **data (IN/OUT)**
OCIAnyData corresponding to the instance. If (*data) is NULL, a new OCIAnyData 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.
**OCIAnyDataSetGetType()**

**Purpose**

Gets the type corresponding to an OCIAnyDataSet.

**Syntax**

```c
sword OCIAnyDataSetGetType ( OCISvcCtx *svchp,
                              OCIError *errhp,
                              OCIAnyDataSet *data_set,
                              OCITypeCode *tc,
                              OCIType **type );
```

**Parameters**

`svchp (IN)`
The OCI service context.

`errhp (IN/OUT)`
The OCI error handle. If there is an error, it is recorded in `err`, and this function returns `OCI_ERROR`. Obtain diagnostic information by calling `OCIErrorGet()`.

`data_set (IN)`
Initialized OCIAnyDataSet.

`tc (OUT)`
The typecode corresponding to the type of the OCIAnyDataSet.

`type (OUT)`
The type corresponding to the OCIAnyDataSet. This is `NULL` if the OCIAnyData corresponds to a built-in type.
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

Introduction to Globalization Support in OCI

This chapter describes the globalization support functions in detail.

See Also: Oracle Database Globalization Support Guide

Conventions for OCI Functions

See the "Conventions for OCI Functions" on page 16-1 for the conventions used in describing each function. 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 Table 22–1.

<table>
<thead>
<tr>
<th>Table 22–1 Function Return Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td>OCI_SUCCESS</td>
</tr>
<tr>
<td>OCI_ERROR</td>
</tr>
<tr>
<td>OCI_INVALID_HANDLE</td>
</tr>
</tbody>
</table>
See Also: "Error Handling in OCI" on page 2-20 for more information about return codes and error handling
OCI Locale Functions

Table 22–2 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>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>“OCINlsCharSetIdToName()” on page 22-4</td>
<td>Return the Oracle Database character set name from the specified character set ID</td>
</tr>
<tr>
<td>“OCINlsCharSetNameToId()” on page 22-5</td>
<td>Return the Oracle Database character set ID for the specified Oracle Database character set name</td>
</tr>
<tr>
<td>“OCINlsEnvironmentVariableGet()” on page 22-6</td>
<td>Return the character set ID from NLS_LANG or the national character set ID from NLS_NCHAR</td>
</tr>
<tr>
<td>“OCINlsGetInfo()” on page 22-8</td>
<td>Copy locale information from an OCI environment or user session handle into an array pointed to by the destination buffer within a specified size</td>
</tr>
<tr>
<td>“OCINlsNumericInfoGet()” on page 22-11</td>
<td>Copy numeric language information from the OCI environment or user session handle into an output number variable</td>
</tr>
</tbody>
</table>
OCINlsCharSetIdToName()

Purpose

Returns the Oracle Database character set name from the specified character set ID.

Syntax

```c
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**.
**OCINlsCharSetNameToId()**

**Purpose**

Returns the Oracle Database character set ID for the specified Oracle Database character set name.

**Syntax**

```c
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.
OCINlsEnvironmentVariableGet()

Purpose

Returns the character set ID from NLS_LANG or the national character set ID from NLS_NCHAR.

Syntax

```c
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
  - OCI_NLS_NCHARSET_ID: NLS_NCHAR character set ID in ub2 data type

- **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

Following globalization support convention, the national character set ID is the same as the character set ID if NLS_NCHAR is not set. If NLS_LANG is not set, then the default character set ID is returned.

To allow for future enhancements of this function (to retrieve other values from environment variables) the 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 and OCI_ATTR_ENV_NCHARSET, respectively.
Returns

OCI_SUCCESS; or OCI_ERROR.

Related Functions

OCIEnvNlsCreate()
OCINlsGetInfo()

Purpose

Obtains locale information from an OCI environment or user session handle to an array pointed to by the destination buffer within a specified size.

Syntax

```c
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_MAXBUFSIZE` 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
OCINLSMONTHNAME1: Native name for January
OCINLSMONTHNAME2: Native name for February
OCINLSMONTHNAME3: Native name for March
OCINLSMONTHNAME4: Native name for April
OCINLSMONTHNAME5: Native name for May
OCINLSMONTHNAME6: Native name for June
OCINLSMONTHNAME7: Native name for July
OCINLSMONTHNAME8: Native name for August
OCINLSMONTHNAME9: Native name for September
OCINLSMONTHNAME10: Native name for October
OCINLSMONTHNAME11: Native name for November
OCINLSMONTHNAME12: Native name for December

OCINLSABMONTHNAME1: Native abbreviated name for January
OCINLSABMONTHNAME2: Native abbreviated name for February
OCINLSABMONTHNAME3: Native abbreviated name for March
OCINLSABMONTHNAME4: Native abbreviated name for April
OCINLSABMONTHNAME5: Native abbreviated name for May
OCINLSABMONTHNAME6: Native abbreviated name for June
OCINLSABMONTHNAME7: Native abbreviated name for July
OCINLSABMONTHNAME8: Native abbreviated name for August
OCINLSABMONTHNAME9: Native abbreviated name for September
OCINLSABMONTHNAME10: Native abbreviated name for October
OCINLSABMONTHNAME11: Native abbreviated name for November
OCINLSABMONTHNAME12: Native abbreviated name for December

OCINLSYES: Native string for affirmative response
OCINLSNO: Native negative response
OCINLSAM: Native equivalent string of AM
OCINLSPM: Native equivalent string of PM
OCINLSAD: Native equivalent string of AD
OCINLSBC: Native equivalent string of BC
OCINLSDECIMAL: Decimal character
OCINGROUP: Group separator
OCINLSDEBIT: Native symbol of debit
OCINLS CREDIT: Native symbol of credit
OCINLSDECFORMAT: Oracle Database date format
OCINLSINTCURRENCY: International currency symbol
OCINLSDUALCURRENCY: Dual currency symbol
OCINLSLOC CURRENCY: Locale currency symbol
OCINLSLANGUAGE: Language name
OCINLSABLANGUAGE: Abbreviation for language name
OCINLS TERRITORY: Territory name
OCINLSCHARACTER_SET: Character set name
OCINLSLINGUISTIC NAME: Linguistic sort name
OCINLS CALENDAR: Calendar name
OCINLSWRITING_DIR: Language writing direction
OCINLSABTERRITORY: Territory abbreviation
OCINLSDDCFORMAT: Oracle Database default date format
OCINLSDTIMEFORMAT: Oracle Database default time format
OCINLSSPDATEFORMAT: Local date format
OCINLS SPTIMEFORMAT: Local time format
OCINLSNUMGROUPING: Number grouping fields
OCINLISTSEP: List separator
OCINLSMONDECIMAL: Monetary decimal character
OCINLSMONGROUP: Monetary group separator
OCINLSMONGROUPING: Monetary grouping fields
OCI_NLS_INT_CURRENCYSEP: International currency separator

Returns

OCI_SUCCESS; OCI_INVALID_HANDLE; or OCI_ERROR.
OCINlsNumericInfoGet()

Purpose
Obtains numeric language information from the OCI environment or user session handle and puts it into an output number variable.

Syntax

```c
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.
OCI Locale-Mapping Function

Table 22–3 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>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCINlsNameMap()&quot; on page 22-13</td>
<td>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</td>
</tr>
</tbody>
</table>
OCINlsNameMap()

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.
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 22–4 summarizes the OCI string manipulation functions, which are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIMultiByteInSizeToWideChar()&quot; on page 22-16</td>
<td>Convert part of a multibyte string into the wide-character string</td>
</tr>
<tr>
<td>&quot;OCIMultiByteStrCaseConversion()&quot; on page 22-17</td>
<td>Convert a multibyte string into the specified case and copies the result into the destination array</td>
</tr>
<tr>
<td>&quot;OCIMultiByteStrcat()&quot; on page 22-18</td>
<td>Append a multibyte string to the destination string</td>
</tr>
<tr>
<td>&quot;OCIMultiByteStrcmp()&quot; on page 22-19</td>
<td>Compare two multibyte strings by binary, linguistic, or case-insensitive comparison methods</td>
</tr>
<tr>
<td>&quot;OCIMultiByteStrcpy()&quot; on page 22-20</td>
<td>Copy a multibyte string into the destination array. It returns the number of bytes copied.</td>
</tr>
<tr>
<td>&quot;OCIMultiByteStrlen()&quot; on page 22-21</td>
<td>Return the number of bytes in a multibyte string</td>
</tr>
<tr>
<td>&quot;OCIMultiByteStrncat()&quot; on page 22-22</td>
<td>Append, at most, ( n ) bytes from a multibyte string to the destination string</td>
</tr>
<tr>
<td>&quot;OCIMultiByteStrncmp()&quot; on page 22-23</td>
<td>Compare two multibyte strings by binary, linguistic, or case-insensitive comparison methods. Each string is in the specified length</td>
</tr>
<tr>
<td>&quot;OCIMultiByteStrncpy()&quot; on page 22-25</td>
<td>Copy a specified number of bytes of a multibyte string into the destination array</td>
</tr>
<tr>
<td>Function</td>
<td>Purpose</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td><code>OCIStringDisplayLength()</code> on page 22-26</td>
<td>Return the number of display positions occupied by the multibyte string within the range of n bytes</td>
</tr>
<tr>
<td><code>OCIStringToWideChar()</code> on page 22-27</td>
<td>Convert a NULL-terminated multibyte string into wide-character format</td>
</tr>
<tr>
<td><code>OCIWideCharInSizeToMultiByte()</code> on page 22-28</td>
<td>Convert part of a wide-character string to the multibyte string</td>
</tr>
<tr>
<td><code>OCIWideCharMultiByteLength()</code> on page 22-29</td>
<td>Determine the number of bytes required for a wide character in multibyte encoding</td>
</tr>
<tr>
<td><code>OCIWideCharStrCaseConversion()</code> on page 22-30</td>
<td>Convert a wide-character string into the specified case and copies the result into the destination array</td>
</tr>
<tr>
<td><code>OCIWideCharStrcat()</code> on page 22-31</td>
<td>Append a wide-character string to the destination string</td>
</tr>
<tr>
<td><code>OCIWideCharStrchr()</code> on page 22-32</td>
<td>Search for the first occurrence of a wide character in a string. Return a point to the wide character if the search is successful.</td>
</tr>
<tr>
<td><code>OCIWideCharStrcmp()</code> on page 22-33</td>
<td>Compare two wide-character strings by binary, linguistic, or case-insensitive comparison methods</td>
</tr>
<tr>
<td><code>OCIWideCharStrcpy()</code> on page 22-34</td>
<td>Copy a wide-character string into a destination array. Return the number of characters copied.</td>
</tr>
<tr>
<td><code>OCIWideCharStrlen()</code> on page 22-35</td>
<td>Return the number of characters in a wide-character string</td>
</tr>
<tr>
<td><code>OCIWideCharStrnca()</code> on page 22-36</td>
<td>Append, at most, n characters from a wide-character string to the destination string</td>
</tr>
<tr>
<td><code>OCIWideCharStrncmp()</code> on page 22-37</td>
<td>Compare two wide-character strings by binary, linguistic, or case-insensitive methods. Each string is a specified length.</td>
</tr>
<tr>
<td><code>OCIWideCharStrncpy()</code> on page 22-39</td>
<td>Copy, at most, n characters of a wide-character string into the destination array</td>
</tr>
<tr>
<td><code>OCIWideCharStrrchr()</code> on page 22-40</td>
<td>Search for the last occurrence of a character in a wide-character string</td>
</tr>
<tr>
<td><code>OCIWideCharToLower()</code> on page 22-41</td>
<td>Convert a specified wide character into the corresponding lowercase character</td>
</tr>
<tr>
<td><code>OCIWideCharToUpper()</code> on page 22-42</td>
<td>Convert a NULL-terminated wide-character string into a multibyte string</td>
</tr>
<tr>
<td><code>OCIWideCharToUpper()</code> on page 22-43</td>
<td>Convert a specified wide character into the corresponding uppercase character</td>
</tr>
</tbody>
</table>
OCIMultiByteInSizeToWideChar()

Purpose

Converts part of a multibyte string into the wide-character string.

Syntax

```c
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 Functions

OCIMultiByteToWideChar()
OCIMultiByteStrCaseConversion()

Purpose
Converts the multibyte string pointed to by srcstr into uppercase or lowercase as specified by the flag and copies the result into the array pointed to by dststr.

Syntax
size_t OCIMultiByteStrCaseConversion ( 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.
OCIMultiByteStrcat()

Purpose

Appends a copy of the multibyte string pointed to by srcstr to the end of the string pointed to by dststr.

Syntax

```c
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 Functions

OCIMultiByteStrncat()
OCI MultiByteStrcmp()

Purpose

Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods.

Syntax

```c
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 Functions

- **OCIMultiByteStrncmp()**
OCIMultiByteStrcpy()

Purpose

Copies the multibyte string pointed to by srcstr into the array pointed to by dststr.

Syntax

```c
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 OCIEEnvNlsCreate() function, then this function produces an error.

Returns

The number of bytes copied, not including the NULL terminator.

Related Functions

- OCIMultiByteStrncpy()
OCI MultiByteStrlen()

Purpose
Returns the number of bytes in the multibyte string pointed to by str, not including the NULL terminator.

Syntax

```c
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 Functions

OCI WideCharStrlen()
OCIMultiByteStrncat()

Purpose

Appends a specified number of bytes from a multibyte string to a destination string.

Syntax

```c
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 Functions

- OCIMultiByteStrcat()
OCIMultiByteStrncmp()

Purpose

Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods. A length is specified for each string.

Syntax

```c
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 Functions

OCIMultiByteStrncpy(), OCIMultiByteStrncpy()
OCIMultiByteStrncpy()

Purpose
Copies a multibyte string into an array.

Syntax
```c
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 Functions
- OCIMultiByteStrcpy()
OCIMultiByteStrnDisplayLength()

Purpose

Returns the number of display positions occupied by the multibyte string within the range of n bytes.

Syntax

```c
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.
OCI MultiByteToWideChar()

Purpose
Converts an entire NULL-terminated string into the wide-character string.

Syntax
```c
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 Functions
OCIWideCharToMultiByte()
OCIWideCharInSizeToMultiByte()

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

- **hdl (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.
OCIWideCharMultiByteLength()

Purpose
Determines the number of bytes required for a wide character in multibyte encoding.

Syntax
```c
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 OCIEnvNlsCreate() function, then this function produces an error.

Returns
The number of bytes required for the wide character.
OCIWideCharStrCaseConversion()

Purpose

Converts a wide-character string into a specified case and copies the result into the destination array.

Syntax

```c
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.
**OCIWideCharStrcat()**

**Purpose**

Appends the wide-character string pointed to by `wsrcstr` to the wide-character string pointed to by `wdststr`.

**Syntax**

```c
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 Functions**

OCIWideCharStrncat()
OCIWideCharStrchr()

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 Functions

OCIWideCharStrrchr()
OCIWideCharStrcmp()

Purpose

Compares two wide-character strings by binary (based on wchar encoding value), linguistic, or case-insensitive comparison methods.

Syntax

```c
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.
  - **OCI_NLS_CASE_INSENSITIVE**: Case-insensitive comparison.

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 Functions

**OCIWideCharStrncmp()**
OCIWideCharStrcpy()

Purpose

Copies a wide-character string into an array.

Syntax

```c
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 Functions

- **OCIWideCharStrncpy()**
**OCIWideCharStrlen()**

**Purpose**

Returns the number of characters in a wide-character string.

**Syntax**

```c
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.
OCIWideCharStrncat()

Purpose

Appends, at most, \( n \) characters from a wide-character string to the destination.

Syntax

```c
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 Functions

OCIWideCharStrcat()
OCIWideCharStrncmp()

Purpose

Compares two wide-character strings by binary, linguistic, or case-sensitive methods. Each string has a specified length.

Syntax

```c
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:
OCIWideCharStrncmp()

- 0, if wstr1 = wstr2
- Positive, if wstr1 > wstr2
- Negative, if wstr1 < wstr2

Related Functions

OCIWideCharStrcmp()
OCIWideCharStrncpy()

Purpose

Copies, at most, \( n \) characters from a wide-character string into a destination.

Syntax

```c
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 \texttt{wchar} buffer.

- **wsrcstr (IN)**
  Pointer to the source \texttt{wchar} string.

- **n (IN)**
  Number of characters from \texttt{wsrcstr} to copy.

Comments

This function is similar to OCIWideCharStrncpy(), except that, at most, \( n \) characters are copied from the array pointed to by \texttt{wsrcstr} to the array pointed to by \texttt{wdststr}. Note that the NULL terminator in \texttt{wdststr} stops copying, and the result string is NULL-terminated. If OCI_UTF16ID is specified for SQL \texttt{CHAR} data in the OCIEnvNlsCreate() function, then this function produces an error.

Returns

The number of characters copied, not including the NULL terminator.

Related Functions

OCIWideCharStrcpy()
OCIWideCharStrrchr()

Purpose

Searches for the last occurrence of a character in a wide-character string.

Syntax

OCIWchar *OCIWideCharStrrchr ( 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 Functions

OCIWideCharStrchr()
OCIWideCharToLower()

Purpose
Converts the wide-character string specified by wc into the corresponding lowercase character, if it exists, in the specified locale. If no corresponding lowercase character exists, then it returns wc itself.

Syntax
OCIWchar OCIWideCharToLower ( 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 Functions
OCIWideCharToUpper()
**OCIWideCharToMultiByte()**

**Purpose**

Converts an entire NULL-terminated wide-character string into a multibyte string.

**Syntax**

```c
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 Functions**

OCIMultiByteToWideChar()
OCIWideCharToUpper()

Purpose

Converts the wide-character string specified by wc into the corresponding uppercase character, if it exists, in the specified locale. If no corresponding uppercase character exists, then it returns wc itself.

Syntax

OCIWchar OCIWideCharToUpper ( void *hndl,
                              OCIWchar wc );

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

cw (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 Functions

OCIWideCharToLower()
**OCI Character Classification Functions**

Table 22–5 lists the OCI character classification functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIWideCharIsAlnum()“ on page 22-45</td>
<td>Test whether the wide character is a letter or a decimal digit</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsAlpha()“ on page 22-46</td>
<td>Test whether the wide character is a letter</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsCtrl()“ on page 22-47</td>
<td>Test whether the wide character is a control character</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsDigit()“ on page 22-48</td>
<td>Test whether the wide character is a decimal digital character</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsGraph()“ on page 22-49</td>
<td>Test whether the wide character is a graph character</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsLower()“ on page 22-50</td>
<td>Test whether the wide character is a lowercase character</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsPrint()“ on page 22-51</td>
<td>Test whether the wide character is a printable character</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsPunct()“ on page 22-52</td>
<td>Test whether the wide character is a punctuation character</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsSingleByte()“ on page 22-53</td>
<td>Test whether the wide character is a single-byte character when converted to multibyte</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsSpace()“ on page 22-54</td>
<td>Test whether the wide character is a space character</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsUpper()“ on page 22-55</td>
<td>Test whether the wide character is an uppercase character</td>
</tr>
<tr>
<td>&quot;OCIWideCharIsXdigit()“ on page 22-56</td>
<td>Test whether the wide character is a hexadecimal digit</td>
</tr>
</tbody>
</table>
OCIWideCharIsAlnum()

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.
OCIWideCharIsAlpha()

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.
OCIWideCharIsCntrl()

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.
OCIWideCharIsDigit()

Purpose
Tests whether a wide character is a decimal digit character.

Syntax

```c
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.
OCIWideCharIsGraph()  

**Purpose**  
Tests whether a wide character is a graph character. A graph character is a character with a visible representation and normally includes alphabetic letters, decimal digits, and punctuation.

**Syntax**  

```c
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.
OCIWideCharIsLower()

Purpose

Tests whether a wide character is a lowercase letter.

Syntax

```c
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`.
OCIWideCharIsPrint()

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.
OCIWideCharIsPunct()

Purpose
Tests whether a wide character is a punctuation character.

Syntax

```c
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.
OCIWideCharIsSingleByte()

Purpose
Tests whether a wide character is a single-byte character when converted to multibyte.

Syntax
boolean OCIWideCharIsSingleByte ( void* hndl,
                                      OCINchar 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.
OCIWideCharIsSpace()

Purpose
Tests whether a wide character is a space character. A space character causes white space only in displayed text (for example, space, tab, carriage return, new line, vertical tab, or form feed).

Syntax

```c
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.
OCIWideCharIsUpper()

Purpose
Tests whether a wide character is an uppercase letter.

Syntax

```c
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.
OCIWideCharIsXdigit()

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.
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 22–6 lists the OCI character set conversion functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCICharSetConversionIsReplacementUsed()&quot; on page 22-58</td>
<td>Indicate whether replacement characters were used for characters that could not be converted in the last invocation of OCINlsCharSetConvert() or OCICharSetToUnicode()</td>
</tr>
<tr>
<td>&quot;OCICharSetToUnicode()&quot; on page 22-59</td>
<td>Convert a multibyte string to Unicode</td>
</tr>
<tr>
<td>&quot;OCINlsCharSetConvert()&quot; on page 22-60</td>
<td>Convert a string from one character set to another</td>
</tr>
<tr>
<td>&quot;OCIUnicodeToCharSet()&quot; on page 22-62</td>
<td>Convert a Unicode string into multibyte</td>
</tr>
</tbody>
</table>
OCICharSetConversionIsReplacementUsed()

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.
OCI CharSetToUnicode()

Purpose
Converts a multibyte string pointed to by src to Unicode out to the array pointed to by dst.

Syntax
sword OCI CharSetToUnicode ( 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.
OCINlsCharSetConvert()

Purpose

Converts a string pointed to by src in the character set specified by srcid to the array pointed to by dst in the character set specified by dstid. The conversion stops when it reaches the data size limitation of either the source or the destination. The function returns the number of bytes converted into the destination buffer.

Syntax

```c
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.
**OCIUnicodeToCharSet()**

**Purpose**

Converts a Unicode string to a multibyte string out to an array.

**Syntax**

```c
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`
- `OCI_ERROR`
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 22–7 lists the OCI messaging functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIMessageClose()&quot; on page 22-64</td>
<td>Close a message handle and free any memory associated with the handle</td>
</tr>
<tr>
<td>&quot;OCIMessageGet()&quot; on page 22-65</td>
<td>Retrieve a message. If the buffer is not zero, then the function copies the message into the buffer</td>
</tr>
<tr>
<td>&quot;OCIMessageOpen()&quot; on page 22-66</td>
<td>Open a message handle in a specified language</td>
</tr>
</tbody>
</table>
OCIMessageClose()

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().

msg (IN/OUT)
A pointer to a message handle that was previously opened by OCIMessageOpen().

Returns

OCI_SUCCESS; OCI_INVALID_HANDLE; or OCI_ERROR.
OCI Messaging Functions

OCIMessageGet()

Purpose
Gets a message with the given message number.

Syntax
```c
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.
OCIMessageOpen()

Purpose

Opens a message-handling facility in a specified language.

Syntax

```c
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.
This chapter describes the OCI XML DB functions.

This chapter contains these topics:

- Introduction to XML DB Support in OCI
- OCI XML DB Functions

## Introduction to XML DB Support in OCI

This chapter describes the XML DB functions in detail.

See Also: "OCI Support for XML" on page 14-17

## Conventions for OCI Functions

See the "Conventions for OCI Functions" on page 16-1 for the conventions used in describing each function. The entries for each function may also contain the following information:

## Returns

Unless otherwise stated, the function returns the values described in Table 23–1.

### Table 23–1 Function Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCI_SUCCESS</td>
<td>The operation succeeded.</td>
</tr>
<tr>
<td>OCI_ERROR</td>
<td>The operation failed. The specific error can be retrieved by calling OCIErrorGet() on the error handle passed to the function.</td>
</tr>
<tr>
<td>OCI_INVALID_HANDLE</td>
<td>The OCI handle passed to the function is invalid.</td>
</tr>
</tbody>
</table>

See Also: "Error Handling in OCI" on page 2-20 for more information about return codes and error handling
Table 23–2 lists the OCI XML DB functions that are described in this chapter.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OCIBinXmlCreate ReposCtxFromConn()&quot; on page 23-3</td>
<td>Create metadata connection context</td>
</tr>
<tr>
<td>&quot;OCIBinXmlCreate ReposCtxFromCPool()&quot; on page 23-4</td>
<td>Create metadata connection context connection pool</td>
</tr>
<tr>
<td>&quot;OCIBinXmlSetFormat Pref()&quot; on page 23-5</td>
<td>Specify that images transferred are in binary XML format</td>
</tr>
<tr>
<td>&quot;OCIBinXmlSet ReposCtxForConn()&quot; on page 23-6</td>
<td>Associate data connection with the metadata connection</td>
</tr>
<tr>
<td>&quot;OCIXmlDbFreeXmlCtx()&quot; on page 23-7</td>
<td>Free an XML context</td>
</tr>
<tr>
<td>&quot;OCIXmlDbInitXmlCtx()&quot; on page 23-8</td>
<td>Initialize an XML context for XML data from the database</td>
</tr>
</tbody>
</table>
**OCIBinXmlCreateReposCtxFromConn()**

**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**
```c
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 Functions**
- OCIBinXmlCreateReposCtxFromCPool()
OCIBinXmlCreateReposCtxFromCPool()

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

```c
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 Functions

- OCIBinXmlCreateReposCtxFromConn()
OCIBinXmlSetFormatPref()

Purpose
 Specifies that the images being transferred between client and server for the XML
document be in binary XML format. 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 Functions
 OCIBinXmlSetReposCtxForConn()
OCIBinXmlSetReposCtxForConn()

Purpose

Associates the data connection with the metadata connection. Note that with a dedicated connection, the environment handle must be the same for the data connection and for the metadata connection.

Syntax

```c
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 Functions

- OCIBinXmlSetFormatPref()
OCI XML DB Functions

OCIXmlDbFreeXmlCtx()

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" on page 14-18 for a usage example

Returns

Returns -1 for error, 0 for success.

Related Functions

OCIXmlDbInitXmlCtx()
OCIXmlDbInitXmlCtx()

Purpose

Initializes an XML context for XML data from the database.

Syntax

```c
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:

  ```c
  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" on page 14-18 for a usage example

Returns

Returns either:

- A pointer to structure `xmlctx`, with error handler and callbacks populated with appropriate values. This is later used for all OCI calls.

- NULL, if no database connection is available.
Related Functions

OCIXmlAttribute()}
This appendix describes the attributes for OCI handles and descriptors, which can be read with `OCIAttrGet()`, and modified with `OCIAttrSet()`. This appendix contains these topics:

- Conventions
- Environment Handle Attributes
- Error Handle Attributes
- Service Context Handle Attributes
- Server Handle Attributes
  - Authentication Information Handle
  - 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
- LOB Locator Attributes
- Complex Object Attributes
  - Complex Object Retrieval Handle Attributes
  - Complex Object Retrieval Descriptor Attributes
- Streams Advanced Queuing Descriptor Attributes
  - `OCIAQEnqOptions` Descriptor Attributes
  - `OCIAQDeqOptions` Descriptor Attributes
  - `OCIAQMgProperties` Descriptor Attributes
  - `OCIAQAgent` Descriptor Attributes
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.

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" on page 14-4 and "Flushing Changes to the Server" on page 14-8

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" on page 14-4
Environment Handle Attributes

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" on page 14-4

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_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_EVTCKB

Mode
WRITE

Description
This attribute registers an event callback function.

See Also: "HA Event Notification" on page 9-43

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" on page 9-43

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:  "Creating Objects" on page 11-23

Attribute Data Type
boolean */boolean

OCI_ATTR_OBJECT_DETECCHANGE

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:  "Implementing Optimistic Locking" on page 14-11

Attribute Data Type
boolean */boolean

OCI_ATTR_PIN_DURATION

Mode
READ/WRITE
Error Handle Attributes

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:

```c
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_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 *

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 *

Service Context Handle Attributes

The following attributes are used for service context handle.
**OCI_ATTR_ENV**

**Mode**  
READ

**Description**  
Returns the environment context associated with the service context.

**Attribute Data Type**  
OCIEnv **

**OCI_ATTR_IN_V8_MODE**

**Mode**  
READ

**Description**  
Allows you to determine whether an application has switched to Oracle release 7 mode (for example, through an OCISvcCtxToLda() call). A nonzero (TRUE) return value indicates that the application is currently running in Oracle release 8 mode, a zero (false) return value indicates that the application is currently running in Oracle release 7 mode.

**Attribute Data Type**  
ub1 *

**Example**  
The following code sample shows how this attribute is used:

```c
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_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**

OCIHandle **/ OCIHandle *

**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:

```c
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**

OCIHandle **/ OCIHandle *
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" on page A-77
- "OCI_ATTR_DBNAME" on page A-77
- "OCI_ATTR_INSTNAME" on page A-79
- "OCI_ATTR_INSTSTARTTIME" on page A-79
- "OCI_ATTR_SERVICENAME" on page A-79

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" on page 9-37

**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_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
Server Handle Attributes

**Handle and Descriptor Attributes**

**A-13**

**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.

**See Also:** "Nonblocking Mode in OCI" on page 2-27

**Attribute Data Type**

`ub1 */ub1`

**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" on page 8-7

**Attribute Data Type**

`oratext **/oratext *`

**OCI_ATTR_SERVER_STATUS**

**Mode**

READ

**Description**

Returns the current status of the server handle. Values are:

- **OCI_SERVER_NORMAL** - There is an active connection to the server. The last call on the connection went through. There is no guarantee that the next call will go through.
- **OCI_SERVER_NOT_CONNECTED** - There is no connection to the server.

**Attribute Data Type**

`ub4 *`

**Example**

The following code sample shows how this parameter is used:

```c
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" on page 9-46

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" on page 9-46

Attribute Data Type
void *

Authentication Information Handle

These attributes also apply to the user session handle.

See Also: "User Session Handle Attributes" on page A-14

User Session Handle Attributes

These attributes also apply to the authentication information handle.

OCI_ATTR_ACTION

Mode
WRITE

Description
The name of the current action within the current module. Can be set to NULL. When the current action terminates, set this attribute again with the name of the next action, or NULL if there is no next action. Can be up to 32 bytes long.

Attribute Data Type
oratext *

Example
OCIAttrSet(session, OCI_HTYPE_SESSION,(void *)"insert into employees",
(ub4)strlen("insert into employees"), OCI_ATTR_ACTION, error_handle);

Note: This attribute is not supported with database resident connection pooling.

Note: This attribute is not supported with database resident connection pooling.
Mode
WRITE

Description
Specifies an attribute name of the externally initialized context.

Attribute Data Type
oratext *

See Also:  "Session Handle Attributes Used to Set an Externally Initialized Context" on page 8-16

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" on page 8-16

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" on page 8-16

OCI_ATTR_AUDIT_BANNER

Mode
READ

Description
Display a user actions auditing banner from a file.

Attribute Data Type
oratext **

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.

Note: This attribute is not supported with database resident connection pooling.

See Also: "Session Handle Attributes Used to Set an Externally Initialized Context" on page 8-16
Server Handle Attributes

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. 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

Client application additional 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 *
Handle and Descriptor Attributes A-17


**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 10g 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_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 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 "ORAS" 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 only ensures that the driver name array is not greater than 30 characters. If more than 9 characters are passed, only the first 8 characters are displayed.

**Attribute Data Type**

oratext **/oratext *

**Example**

```c
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, 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_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" on page 8-7
**Attribute Data Type**

`ub1` *

**Example**
The following code sample shows how this attribute is used:

```c
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_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
OCIAuthInfo

**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(), the purity value OCI_ATTR_PURITY_DEFAULT is assumed. This later translates to either OCI_ATTR_PURITY_NEW or OCI_ATTR_PURITY_SELF depending on the type of application.

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.

See Also: "Marking Sessions Explicitly as Stateful or Stateless" on page 9-21 for more information and an example using OCI_ATTR_SESSION_STATE

Attribute Data Type
ub1 *

**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);
}
Administration Handle Attributes

The following attributes are used for the administration handle.

OCI_ATTR_USERNAME

Mode
READ/WRITE

Description
Specifies a user name to use for authentication.

Attribute Data Type
oratext */oratext *

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 *

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 *

Session Pool Handle Attributes
The following attributes are used for the session pool handle.

OCI_ATTR_SPOOL_BUSY_COUNT

Mode
READ

Description
Returns the number of busy sessions.

Attribute Data Type
ub4 *

OCI_ATTR_SPOOL_GETMODE

Mode
READ/WRITE

Description
This attribute determines the behavior of the session pool when all sessions in the pool are found to be busy and the number of sessions has reached the maximum. Values are:

- OCI_SPOOL_ATTRVAL_WAIT - The thread waits and blocks until a session is freed. This is the default value.
- OCI_SPOOL_ATTRVAL_NOWAIT - An error is returned.
- OCI_SPOOL_ATTRVAL_FORCEGET - A new session is created even though all the sessions are busy and the maximum number of sessions has been reached. OCISessionGet() returns a warning. In this case, if new sessions are created that have exceeded the maximum, OCISessionGet() returns a warning.

Note that if this value is set, it is possible that there can be an attempt to create more sessions than can be supported by the instance of the Oracle database. 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_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_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

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_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" on page 9-26

**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.

**Attribute Data Type**

ub4 */ ub4

**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)

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.

Attribute Data Type
OCISubscription *

Example
/* Associate the statement with the subscription handle */
OCIAttrSet (stmthp, OCI_HTYPE_STMT, subscrhp, 0,
   OCI_ATTR_CHNF_REGHANDLE, errhp);

See Also: "Continuous Query Notification Attributes" on page A-57

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
Indicates the current position in the result set. This attribute can only be retrieved. It cannot be set.

Attribute Data Type
ub4 *

OCI_ATTR_FETCH_ROWID

Mode
READ/WRITE
Description
Specifies that the ROWIDs are fetched after doing a define at position 0, and a SELECT...FOR UPDATE statement.

Attribute Data Type
boolean */boolean

See Also: "Implicit Fetching of ROWIDs" on page 10-20

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, \texttt{OCI\_ATTR\_ROW\_COUNT} is the total number of rows fetched into user buffers with the \texttt{OCISqlStmtFetch2()} 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, \texttt{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.

\textbf{Attribute Data Type} \\
\texttt{ub4 *}

\textbf{OCI\_ATTR\_ROW\_ID}

\textbf{Mode} \\
READ

\textbf{Description} \\
Returns the \texttt{ROWID} descriptor allocated with \texttt{OCIDescriptorAlloc()}.

\textbf{See Also:} "Positioned Updates and Deletes" on page 2-25 and "ROWID Descriptor" on page 3-16

\textbf{Attribute Data Type} \\
\texttt{OCIRowid *}

\textbf{OCI\_ATTR\_ROWS\_FETCHED}

\textbf{Mode} \\
READ

\textbf{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.

\textbf{Attribute Data Type} \\
\texttt{ub4 *}

\textbf{Example} \\
\texttt{ub4 rows;}
\texttt{ub4 sizep = sizeof(ub4);}
\texttt{OCIAttrGet((void *) stmhp, (ub4) OCI\_HTYPE\_STMT,}
\texttt{(void *)& rows, (ub4 *) &sizep, (ub4)OCI\_ATTR\_ROWS\_FETCHED, errhp);

\textbf{OCI\_ATTR\_SQLFCN\_CODE}

\textbf{Mode} \\
READ

\textbf{Description} \\
Returns the function code of the SQL command associated with the statement.

\textbf{Attribute Data Type} \\
\texttt{ub2 *}
Notes
Table A–1 lists the SQL command codes.

<table>
<thead>
<tr>
<th>Code</th>
<th>SQL Function</th>
<th>Code</th>
<th>SQL Function</th>
<th>Code</th>
<th>SQL Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>CREATE TABLE</td>
<td>43</td>
<td>DROP EXTERNAL DATABASE</td>
<td>85</td>
<td>TRUNCATE TABLE</td>
</tr>
<tr>
<td>02</td>
<td>SET ROLE</td>
<td>44</td>
<td>CREATE DATABASE</td>
<td>86</td>
<td>TRUNCATE CLUSTER</td>
</tr>
<tr>
<td>03</td>
<td>INSERT</td>
<td>45</td>
<td>ALTER DATABASE</td>
<td>87</td>
<td>CREATE BITMAPFILE</td>
</tr>
<tr>
<td>04</td>
<td>SELECT</td>
<td>46</td>
<td>CREATE ROLLBACK SEGMENT</td>
<td>88</td>
<td>ALTER VIEW</td>
</tr>
<tr>
<td>05</td>
<td>UPDATE</td>
<td>47</td>
<td>ALTER ROLLBACK SEGMENT</td>
<td>89</td>
<td>DROP BITMAPFILE</td>
</tr>
<tr>
<td>06</td>
<td>DROP ROLE</td>
<td>48</td>
<td>DROP ROLLBACK SEGMENT</td>
<td>90</td>
<td>SET CONSTRAINTS</td>
</tr>
<tr>
<td>07</td>
<td>DROP VIEW</td>
<td>49</td>
<td>CREATE TABLESPACE</td>
<td>91</td>
<td>CREATE FUNCTION</td>
</tr>
<tr>
<td>08</td>
<td>DROP TABLE</td>
<td>50</td>
<td>ALTER TABLESPACE</td>
<td>92</td>
<td>ALTER FUNCTION</td>
</tr>
<tr>
<td>09</td>
<td>DELETE</td>
<td>51</td>
<td>DROP TABLESPACE</td>
<td>93</td>
<td>DROP FUNCTION</td>
</tr>
<tr>
<td>10</td>
<td>CREATE VIEW</td>
<td>52</td>
<td>ALTER SESSION</td>
<td>94</td>
<td>CREATE PACKAGE</td>
</tr>
<tr>
<td>11</td>
<td>DROP USER</td>
<td>53</td>
<td>ALTER USER</td>
<td>95</td>
<td>ALTER PACKAGE</td>
</tr>
<tr>
<td>12</td>
<td>CREATE ROLE</td>
<td>54</td>
<td>COMMIT (WORK)</td>
<td>96</td>
<td>DROP PACKAGE</td>
</tr>
<tr>
<td>13</td>
<td>CREATE SEQUENCE</td>
<td>55</td>
<td>ROLLBACK</td>
<td>97</td>
<td>CREATE PACKAGE BODY</td>
</tr>
<tr>
<td>14</td>
<td>ALTER SEQUENCE</td>
<td>56</td>
<td>SAVEPOINT</td>
<td>98</td>
<td>ALTER PACKAGE BODY</td>
</tr>
<tr>
<td>15</td>
<td>(NOT USED)</td>
<td>57</td>
<td>CREATE CONTROL FILE</td>
<td>99</td>
<td>DROP PACKAGE BODY</td>
</tr>
<tr>
<td>16</td>
<td>DROP SEQUENCE</td>
<td>58</td>
<td>ALTER TRACING</td>
<td>157</td>
<td>CREATE DIRECTORY</td>
</tr>
<tr>
<td>17</td>
<td>CREATE SCHEMA</td>
<td>59</td>
<td>CREATE TRIGGER</td>
<td>158</td>
<td>DROP DIRECTORY</td>
</tr>
<tr>
<td>18</td>
<td>CREATE CLUSTER</td>
<td>60</td>
<td>ALTER TRIGGER</td>
<td>159</td>
<td>CREATE LIBRARY</td>
</tr>
<tr>
<td>19</td>
<td>CREATE USER</td>
<td>61</td>
<td>DROP TRIGGER</td>
<td>160</td>
<td>CREATE JAVA</td>
</tr>
<tr>
<td>20</td>
<td>CREATE INDEX</td>
<td>62</td>
<td>ANALYZE TABLE</td>
<td>161</td>
<td>ALTER JAVA</td>
</tr>
<tr>
<td>21</td>
<td>DROP INDEX</td>
<td>63</td>
<td>ANALYZE INDEX</td>
<td>162</td>
<td>DROP JAVA</td>
</tr>
<tr>
<td>22</td>
<td>DROP CLUSTER</td>
<td>64</td>
<td>ANALYZE CLUSTER</td>
<td>163</td>
<td>CREATE OPERATOR</td>
</tr>
<tr>
<td>23</td>
<td>VALIDATE INDEX</td>
<td>65</td>
<td>CREATE PROFILE</td>
<td>164</td>
<td>CREATE INDEXTYPE</td>
</tr>
<tr>
<td>24</td>
<td>CREATE PROCEDURE</td>
<td>66</td>
<td>DROP PROFILE</td>
<td>165</td>
<td>DROP INDEXTYPE</td>
</tr>
<tr>
<td>25</td>
<td>ALTER PROCEDURE</td>
<td>67</td>
<td>ALTER PROFILE</td>
<td>166</td>
<td>ALTER INDEXTYPE</td>
</tr>
<tr>
<td>26</td>
<td>ALTER TABLE</td>
<td>68</td>
<td>DROP PROCEDURE</td>
<td>167</td>
<td>DROP OPERATOR</td>
</tr>
<tr>
<td>27</td>
<td>EXPLAIN</td>
<td>69</td>
<td>(NOT USED)</td>
<td>168</td>
<td>ASSOCIATE STATISTICS</td>
</tr>
<tr>
<td>28</td>
<td>GRANT</td>
<td>70</td>
<td>ALTER RESOURCE COST</td>
<td>169</td>
<td>DISASSOCIATE STATISTICS</td>
</tr>
<tr>
<td>29</td>
<td>REVOKE</td>
<td>71</td>
<td>CREATE SNAPSHOT LOG</td>
<td>170</td>
<td>CALL METHOD</td>
</tr>
<tr>
<td>30</td>
<td>CREATE SYNONYM</td>
<td>72</td>
<td>ALTER SNAPSHOT LOG</td>
<td>171</td>
<td>CREATE SUMMARY</td>
</tr>
<tr>
<td>31</td>
<td>DROP SYNONYM</td>
<td>73</td>
<td>DROP SNAPSHOT LOG</td>
<td>172</td>
<td>ALTER SUMMARY</td>
</tr>
</tbody>
</table>
### 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_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**

\[\text{ub4 *}\]

**OCI_ATTR_STMT_TYPE**

**Mode**
READ

**Description**
The type of statement associated with the handle. Valid values are:

- OCI_STMT_SELECT
- OCI_STMT_UPDATE
- OCI_STMT_DELETE
- OCI_STMT_INSERT
- OCI_STMT_CREATE
- OCI_STMT_DROP
- OCI_STMT.Alter
- OCI_STMT.Begin (PL/SQL statement)
- OCI_STMT.Declare (PL/SQL statement)

**Attribute Data Type**

\[\text{ub2 *}\]

---

**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" on page 5-29

**Attribute Data Type**

\[\text{ub4 *}\]
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 (replaces the deprecated OCI_UC2SID, which is retained for backward compatibility), the user must set the character set ID to OCI_UTF16ID. 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" on page 5-26

Attribute Data Type
ub2 *

OCI_ATTR_MAXCHAR_SIZE

Mode
WRITE

Description
Sets the number of characters that an application reserves on the server to store the data being bound.

See Also: "Using the OCI_ATTR_MAXCHAR_SIZE Attribute" on page 5-29

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: "Using the OCI_ATTR_MAXDATA_SIZE Attribute" on page 5-28

Attribute Data Type
sb4 *

OCI_ATTR_PDPRC

Mode
WRITE

Description
Specifies packed decimal precision. For SQLT_PDN values, the precision should be equal to \(2 \times (\text{value}_\text{sz} - 1)\). For SQLT_SLS values, the precision should be equal to \((\text{value}_\text{sz} - 1)\).

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 *
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” on page 5-26

**Attribute Data Type**
ub2 *
**OCI_ATTR_LOBPREFETCH_LENGTH**

**Mode**
READ/WRITE

**Description**
Specifies the prefetch length and chunk size for the LOB locators to be fetched from a particular column.

**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.

**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:**  "Using the OCI_ATTR_MAXCHAR_SIZE Attribute" on page 5-29

**Attribute Data Type**
sb4 *

**OCI_ATTR_PDPRC**

**Mode**
WRITE

**Description**
Specifies packed decimal precision. For SQLT_PDN values, the precision should be equal to \(2^{\text{value}\_sz-1}\). For SQLT_SLS values, the precision should be equal to \(\text{value}\_sz-1\).

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 *
LOB Locator Attributes

**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 *`

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 *`

Parameter Descriptor Attributes

The following attributes are used for the parameter descriptor.

For a detailed list of parameter descriptor attributes, see Chapter 6.

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:

```c
ub4 lobEmpty = 0
```

Then they should pass the address: &lobEmpty.

**Attribute Data Type**

```c
ub4 *
```

---

**Complex Object Attributes**

The following attributes are used for complex objects.

**See Also:** "Complex Object Retrieval" on page 11-15

---

**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**

```c
ub4 *
```

---

**OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE**

**Mode**
WRITE

**Description**
Whether to fetch collection attributes in an object type out-of-line.

**Attribute Data Type**

```c
ub1 *
```

---

**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 REFs of type OCI_ATTR_COMPLEXOBJECTCOMP_TYPE.

**Attribute Data Type**
ub4 *

---

**Streams Advanced Queuing Descriptor Attributes**

The following attributes are used for the streams advanced queuing descriptor.

**See Also:** Oracle Streams Advanced Queuing User’s Guide

**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.
**O CIAQDeqOptions Descriptor Attributes**

The following attributes are properties of the `O CIAQDeqOptions` 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 OCIAQDepOptions 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.
**Streams Advanced Queuing Descriptor Attributes**

**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.

**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*
OCI_Date

**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 *

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.

OCI ServerDNs Descriptor Attributes

The following attributes are properties of the OCI ServerDNs 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 *

Subscription Handle Attributes

The following attributes are used for the subscription handle.

**See Also:**
- "Publish-Subscribe Notification in OCI" on page 9-54
- "Continuous Query Notification" on page 10-1

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 "Subscription Handle Attributes for Continuous Query Notification" on page 10-3.

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 OCISubscriptionRegister().

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,

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,
(void *) ipaddr, (ub4) strlen(ipaddr),
(void) 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

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_NAMESPACE.

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.
Subscription Handle Attributes

**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 *

---

**Note:** OCI_OBJECT mode is required when using grouping notifications.

**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.

**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_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_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" on page 9-64 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`. 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` *

**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.

Attribute Data Type
ub4 *

See Also: "Continuous Query Notification" on page 10-1 for details about the flag values

**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 **

**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.

Attribute Data Type
ub4 *

See Also: "Continuous Query Notification" on page 10-1 for the flag values

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_CHANGE_DESC 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: "OCI_DTYPE_TABLE_CHDES" on page 10-6 for the flag values

OCI_ATTR_CHDES_TABLE_ROW_CHANGES

Mode
READ

Description
An embedded collection describing the changes to the rows of the table. Each element of the collection is a row 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 **

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
Direct Path Loading Handle Attributes

**Description**
The queue name of the notification.

**Attribute Data Type**
oratext *

Invalidated Query Attributes

This section describes OCI_DTYPE_CQDES attributes. See "OCI_DTYPE_CQDES" on page 10-6 for more information.

**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 *

Direct Path Loading Handle Attributes

The following attributes are used for the direct path loading handle.

See Also:  "Direct Path Loading Overview" on page 13-1 and "Direct Path Loading of Object Types" on page 13-14 for information about direct path loading and allocating the direct path handles.
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:** "Using a Date Cache in Direct Path Loading of Dates in OCI" on page 13-12 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_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
**Direct Path Loading Handle Attributes**

### Handle and Descriptor Attributes

**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

**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.

**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.

**OCI_ATTR_DIRPATH_OBJ_CONSTR**

**Mode**
READ/WRITE

**Description**
Indicates the object type of a substitutable object table:

```c
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);

**Direct Path Loading Handle Attributes**

### 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_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_LIST_COLUMNS

**Mode**

READ

**Description**

Returns the handle to the parameter descriptor for the column list associated with the direct path context. The column list parameter descriptor can be retrieved after the number of columns is set with the OCI_ATTR_NUM_COLS attribute.

**See Also:** "Accessing Column Parameter Attributes" on page A-71

**Attribute Data Type**

- **OCIParam* * 

### OCI_ATTR_NAME

**Mode**

READ/WRITE

---

A-66 Oracle Call Interface Programmer’s Guide
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 *
Direct Path Function Context Handle (OCI DirPathFuncCtx) Attributes

For further explanations of these attributes, see "Direct Path Function Context and Attributes" on page 13-28.

**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:*  "Accessing Column Parameter Attributes" on page A-71

*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 *

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 *
Direct Path Loading Handle Attributes

**OCI_ATTR_ROW_COUNT**

**Mode**
READ

**Description**
Number of rows successfully converted in the last call to OCIDirPathColArrayToStream().

**Attribute Data Type**
ub4 *

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 *

Direct Path Column Parameter Attributes

The application specifies which columns are to be loaded, and the external format of the data by setting attributes on each column parameter descriptor. The column parameter descriptors are obtained as parameters of the column parameter list by OCIParamGet(). The column parameter list of the table is obtained from the OCI_ATTR_LIST_COLUMNS attribute of the direct path context. If a column is 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.

Accessing Column Parameter Attributes

The following code example illustrates the use of the direct path column parameter attributes for scalar columns. Before the attributes are accessed, you must first set the number of columns to be loaded and get the column parameter list from the OCI_ATTR_LIST_COLUMNS attribute.

See Also: "Direct Path Load Examples for Scalar Columns" on page 13-7 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 *)&tblp->ncol_tbl,  
(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));
}

...
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 *

---

**Process Handle Attributes**

The parameters for the shared system can be set and read using the `OCIAttrSet()` and `OCIAttrGet()` calls. The handle type to be used is the process handle `OCI_HTYPE_PROC`.

*See Also*: "OCI_ATTR_SHARED_HEAPALLOC" on page A-8

The `OCI_ATTR_MEMPOOL_APPNAME`, `OCI_ATTR_MEMPOOL_HOMENAME`, and `OCI_ATTR_MEMPOOL_INSTNAME` attributes specify the application, home, and instance names that can be used together to map the process to the right shared pool area. If these attributes are not provided, internal default values are used. The following are valid settings of the attributes for specific behaviors:

- **Instance name, application name (unqualified):** This allows only executables with a specific name to attach to the same shared subsystem. For example, this allows an OCI application named `Office` to connect to the same shared subsystem regardless of the directory `Office` resides in.

- **Instance name, home name:** This allows a set of executables in a specific home directory to attach to the same instance of the shared subsystem. For example, this allows all OCI applications residing in the `ORACLE_HOME` directory to use the same shared subsystem.

- **Instance name, home name, application name (unqualified):** This allows only a specific executable to attach to a shared subsystem. For example, this allows one application named `Office` in the `ORACLE_HOME` directory to attach to a given shared subsystem.

**OCI_ATTR_MEMPOOL_APPNAME**

**Mode**
READ/WRITE

**Description**
Executable name or fully qualified path name of the executable.

*Attribute 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:

```c
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:

```c
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;
```
Event Handle Attributes

The OCIEvent handle encapsulates the attributes from the event payload. This handle is implicitly allocated before the event callback is called.

**See Also:** "HA Event Notification" on page 9-43

The event callback obtains the attributes of an event using `OCIAttrGet()` with the following attributes.

### OCI_ATTR_DBDOMAIN

**Mode**
READ

**Description**
When called with this attribute, `OCIAttrGet()` retrieves the name of the database domain that has been affected by this event. 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:
Event Handle Attributes

- 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 with this 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 **

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 **
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 Appendix D.

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>
<thead>
<tr>
<th>Program Name</th>
<th>Features Illustrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdemo81.c</td>
<td>Using basic SQL processing with release 8 functionality</td>
</tr>
<tr>
<td>cdemo82.c</td>
<td>Performing basic processing of user-defined objects</td>
</tr>
<tr>
<td>cdemocor.c</td>
<td>Using complex object retrieval (COR) to improve performance</td>
</tr>
<tr>
<td>cdemodr1.c</td>
<td>Using INSERT, UPDATE, and DELETE statements, with RETURNING clause used with basic data types, LOBs and REFs</td>
</tr>
<tr>
<td>cdemodr2.c</td>
<td></td>
</tr>
<tr>
<td>cdemodr3.c</td>
<td></td>
</tr>
</tbody>
</table>
**Table B–1 (Cont.) OCI Demonstration Programs**

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Features Illustrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdemodsa.c</td>
<td>Describing information about a table</td>
</tr>
<tr>
<td>cdemodsc.c</td>
<td>Describing information about an object type</td>
</tr>
<tr>
<td>cdemofo.c</td>
<td>Registering and operating application failover callbacks</td>
</tr>
<tr>
<td>cdemolb.c</td>
<td>Creating and inserting LOB data and then reading, writing, coping, appending, and trimming the data</td>
</tr>
<tr>
<td>cdemolb2.c</td>
<td>Writing and reading of CLOB and BLOB columns with stream mode and callback functions</td>
</tr>
<tr>
<td>cdemolbs.c</td>
<td>Writing and reading to LOBs with the LOB buffering system</td>
</tr>
<tr>
<td>cdemobj.c</td>
<td>Pinning and navigation of REF object</td>
</tr>
<tr>
<td>cdemocoll.c</td>
<td>Inserting and selecting of nested table and varray</td>
</tr>
<tr>
<td>cdemorid.c</td>
<td>Using INSERT, UPDATE, DELETE statements and fetches to get multiple rowids in one round-trip</td>
</tr>
<tr>
<td>cdemoses.c</td>
<td>Using session switching and migration</td>
</tr>
<tr>
<td>cdemothr.c</td>
<td>Using the OCIThread package</td>
</tr>
<tr>
<td>cdemosyev.c</td>
<td>Registering predefined subscriptions and specifying a callback function to be invoked for client notifications (for more information about Advanced Queuing, see Oracle Streams Advanced Queuing User's Guide)</td>
</tr>
<tr>
<td>ociaqdemo00.c</td>
<td>Streams Advanced Queuing. Enqueues 100 messages</td>
</tr>
<tr>
<td>ociaqdemo01.c</td>
<td>Dequeues messages by blocking</td>
</tr>
<tr>
<td>ociaqdemo02.c</td>
<td>Listens for multiple agents</td>
</tr>
<tr>
<td>ociaqarrayeq.c</td>
<td>Array enqueue of 10 messages</td>
</tr>
<tr>
<td>ociaqarraydeq.c</td>
<td>Array dequeue of 10 messages</td>
</tr>
<tr>
<td>cdemodp.c,</td>
<td>Loading data with the direct path load functions</td>
</tr>
<tr>
<td>cdemodp_lip.c</td>
<td></td>
</tr>
<tr>
<td>cdemdpcu.c</td>
<td>Loading a column object with the direct path load functions</td>
</tr>
<tr>
<td>cdemdpln.c</td>
<td>Loading a nested column object with the direct path load functions</td>
</tr>
<tr>
<td>cdemdpi.c</td>
<td>Loading derived type (inheritance) - direct path</td>
</tr>
<tr>
<td>cdemdpi2.c</td>
<td>Loading an object table with inheritance - direct path</td>
</tr>
<tr>
<td>cdemdpout.c</td>
<td>Loading a reference with the direct path load functions</td>
</tr>
<tr>
<td>cdemdpss.c</td>
<td>Loading SQL strings with the direct path load functions</td>
</tr>
<tr>
<td>cdemouchb.c,</td>
<td>Using static and dynamic user callbacks</td>
</tr>
<tr>
<td>cdemouclb.c</td>
<td></td>
</tr>
<tr>
<td>cdemoupk.c,</td>
<td>Using dynamic user callbacks with multiple packages</td>
</tr>
<tr>
<td>cdemoup1.c,</td>
<td></td>
</tr>
<tr>
<td>cdemoup2.c</td>
<td></td>
</tr>
<tr>
<td>cdemodt.c</td>
<td>Datetime and interval example. Demonstrates IN and OUT binds with PL/SQL procedure or function</td>
</tr>
<tr>
<td>cdemosc.c</td>
<td>Scrollable cursor</td>
</tr>
<tr>
<td>cdemol21.c</td>
<td>Accesses LOBs using the LONG API (Data Interface)</td>
</tr>
<tr>
<td>cdemoinl1.c</td>
<td>Inheritance demo that modifies an inherited type in a table and displays a record from the table</td>
</tr>
<tr>
<td>cdemoin2.c</td>
<td>Inheritance demo to do attribute substitutability</td>
</tr>
<tr>
<td>Program Name</td>
<td>Features Illustrated</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cdemoin3.c</td>
<td>Inheritance demo that describes an object, inherited types, object tables, and a sub-table</td>
</tr>
<tr>
<td>cdemoanydata1.c</td>
<td>Anydata demo. Inserts and selects rows to and from anydata table</td>
</tr>
<tr>
<td>cdemoanydata2.c</td>
<td>Anydata demo. Creates a type piecewise using OCITypeBeginCreate() and then describes the new type created</td>
</tr>
<tr>
<td>cdemoqc.c</td>
<td>Query caching using SQL hints</td>
</tr>
<tr>
<td>cdemoqc2.c</td>
<td>Query caching using SQL hints and table annotation</td>
</tr>
<tr>
<td>cdemosp.c</td>
<td>Session pooling</td>
</tr>
<tr>
<td>cdemocp.c</td>
<td>Connection pooling</td>
</tr>
<tr>
<td>cdemocproxy.c</td>
<td>Connection pooling with proxy functionality</td>
</tr>
<tr>
<td>cdemostc.c</td>
<td>Statement caching</td>
</tr>
<tr>
<td>cdemouni.c</td>
<td>Program for OCI UTF16 API</td>
</tr>
<tr>
<td>nchdemo1.c</td>
<td>Shows NCHAR implicit conversion feature and code point feature</td>
</tr>
</tbody>
</table>
This appendix provides information about server round-trips incurred during various OCI calls. A server round-trip is defined as the trip from the client to the server and back to the client. This information can help programmers to determine the most efficient way to accomplish a particular task in an application.

This appendix contains these topics:

- Overview of Server Round-Trips
- 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

Overview of Server Round-Trips

This appendix provides information about server round-trips incurred during various OCI calls. This information can be useful when determining the most efficient way to accomplish a particular task in an application.

Relational Function Round-Trips

Table C–1 lists the number of server round-trips required by each OCI relational function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of Server Round-Trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIBreak()</td>
<td>1</td>
</tr>
<tr>
<td>OCIDBShutdown()</td>
<td>1</td>
</tr>
<tr>
<td>OCIDBStartup()</td>
<td>1</td>
</tr>
<tr>
<td>OCIEnvCreate()</td>
<td>0</td>
</tr>
<tr>
<td>OCIEnvInit()</td>
<td>0</td>
</tr>
<tr>
<td>OCIErrorGet()</td>
<td>0</td>
</tr>
</tbody>
</table>
## Table C–1  (Cont.) Server Round-Trips for Relational Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of Server Round-Trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIInitialize()</td>
<td>0</td>
</tr>
<tr>
<td>OCILdaToSvcCtx()</td>
<td>0</td>
</tr>
<tr>
<td>OCILogoff()</td>
<td>1</td>
</tr>
<tr>
<td>OCILogon()</td>
<td>1</td>
</tr>
<tr>
<td>OCILogon2()</td>
<td>1</td>
</tr>
<tr>
<td>OCILogon2()</td>
<td>Connection pool or session pool: same as OCISessionGet()</td>
</tr>
<tr>
<td>OCILogon2()</td>
<td>Normal: 2 (depends on authentication and TAF situation)</td>
</tr>
<tr>
<td>OCILogon2()</td>
<td></td>
</tr>
<tr>
<td>OCIPasswordChange()</td>
<td>1</td>
</tr>
<tr>
<td>OCIPing()</td>
<td>1</td>
</tr>
<tr>
<td>OCIRest()</td>
<td>0</td>
</tr>
<tr>
<td>OCIServerAttach()</td>
<td>1</td>
</tr>
<tr>
<td>OCIServerAttach()</td>
<td></td>
</tr>
<tr>
<td>OCIServerAttach()</td>
<td></td>
</tr>
<tr>
<td>OCIServerDetach()</td>
<td>1</td>
</tr>
<tr>
<td>OCIServerDetach()</td>
<td></td>
</tr>
<tr>
<td>OCIServerVersion()</td>
<td>1</td>
</tr>
<tr>
<td>OCISessionBegin()</td>
<td>1</td>
</tr>
<tr>
<td>OCISessionBegin()</td>
<td></td>
</tr>
<tr>
<td>OCISessionEnd()</td>
<td>1</td>
</tr>
<tr>
<td>OCISessionEnd()</td>
<td></td>
</tr>
<tr>
<td>OCISessionGet()</td>
<td>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.</td>
</tr>
<tr>
<td>OCISessionGet()</td>
<td>Normal: 1 login</td>
</tr>
<tr>
<td>OCISessionPoolCreate()</td>
<td>sessMin * logins</td>
</tr>
<tr>
<td>OCISessionPoolDestroy()</td>
<td>Sessions in cache * logoffs</td>
</tr>
<tr>
<td>OCISessionRelease()</td>
<td>Session pooling: 0, except when explicit session destroys flag set</td>
</tr>
<tr>
<td>OCISessionRelease()</td>
<td>Normal: 1 login</td>
</tr>
<tr>
<td>OCISemExecute()</td>
<td>1</td>
</tr>
<tr>
<td>OCISemFetch()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCISemFetch2()</td>
<td>0 in prefetch, otherwise 1</td>
</tr>
<tr>
<td>OCISemGetPieceInfo()</td>
<td>1</td>
</tr>
<tr>
<td>OCISemPrepare()</td>
<td>0</td>
</tr>
<tr>
<td>OCISemSetPieceInfo()</td>
<td>0</td>
</tr>
<tr>
<td>OCISvcCtxToLda()</td>
<td>0</td>
</tr>
<tr>
<td>OCITerminate()</td>
<td>1</td>
</tr>
<tr>
<td>OCITransCommit()</td>
<td>1</td>
</tr>
<tr>
<td>OCITransCommit()</td>
<td></td>
</tr>
<tr>
<td>OCITransDetach()</td>
<td>1</td>
</tr>
<tr>
<td>OCITransDetach()</td>
<td></td>
</tr>
<tr>
<td>OCITransForget()</td>
<td>1</td>
</tr>
<tr>
<td>OCITransForget()</td>
<td></td>
</tr>
<tr>
<td>OCITransPrepare()</td>
<td>1</td>
</tr>
<tr>
<td>OCITransPrepare()</td>
<td></td>
</tr>
<tr>
<td>OCITransRollback()</td>
<td>1</td>
</tr>
<tr>
<td>OCITransRollback()</td>
<td></td>
</tr>
</tbody>
</table>
Table C–2 lists the server round-trips incurred by the OCILob calls.

**Note:** To minimize the number of round-trips, you can use the data interface for LOBs. You can bind or define character data for a CLOB column or RAW data for a BLOB column.

See Also:
- "Binding LOB Data" on page 5-9 for usage and examples for both INSERT and UPDATE statements
- "Defining LOB Data" on page 5-16 for usage and examples of SELECT statements

For calls whose number of round-trips is "0 or 1," if LOB buffering is on, and the request can be satisfied in the client, no round-trips are incurred.

### Table C–2 Server Round-Trips for OCILob Calls

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of Server Round-Trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCILobAppend()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobArrayRead()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobArrayWrite()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobAssign()</td>
<td>0</td>
</tr>
<tr>
<td>OCILobCharSetForm()</td>
<td>0</td>
</tr>
<tr>
<td>OCILobCharSetId()</td>
<td>0</td>
</tr>
<tr>
<td>OCILobClose()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobCopy()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobCopy2()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobCreateTemporary()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobDisableBuffering()</td>
<td>0</td>
</tr>
<tr>
<td>OCILobEnableBuffering()</td>
<td>0</td>
</tr>
<tr>
<td>OCILobErase()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobErase2()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobFileClose()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobFileCloseAll()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobFileExists()</td>
<td>1</td>
</tr>
<tr>
<td>OCILobFileName()</td>
<td>0</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Function</th>
<th>Number of Server Round-Trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCICacheFlush()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheFree()</td>
<td>0</td>
</tr>
<tr>
<td>OCICacheRefresh()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheUnmark()</td>
<td>0</td>
</tr>
<tr>
<td>OCICacheUnpin()</td>
<td>0</td>
</tr>
<tr>
<td>OCICacheFlush()</td>
<td>1 for each modified page in the buffer for this LOB</td>
</tr>
<tr>
<td>OCICacheFreeTemporary()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheGetChunkSize()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheGetLength()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheGetLength2()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheGetStorageLimit()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheIsEqual()</td>
<td>0</td>
</tr>
<tr>
<td>OCICacheIsOpen()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheIsTemporary()</td>
<td>0</td>
</tr>
<tr>
<td>OCICacheLoadFromFile()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheLoadFromFile2()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheLocatorAssign()</td>
<td>1 round-trip if either the source or the destination locator refers to a temporary LOB</td>
</tr>
<tr>
<td>OCICacheLocatorIsInit()</td>
<td>0</td>
</tr>
<tr>
<td>OCICacheOpen()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheRead()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCICacheRead2()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCICacheTrim()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheTrim2()</td>
<td>1</td>
</tr>
<tr>
<td>OCICacheWrite()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCICacheWrite2()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCICacheWriteAppend()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCICacheWriteAppend2()</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

Object and Cache Function Round-Trips

Table C–3 lists the number of server round-trips required for the object and cache functions. These values assume the cache is in a warm state, meaning that the type descriptor objects required by the application have been loaded.

Table C–3  Server Round-Trips for Object and Cache Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of Server Round-Trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIObjectIsOpen()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectOpen()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectSetName()</td>
<td>0</td>
</tr>
<tr>
<td>OCIObjectFlushBuffer()</td>
<td>1 for each modified page in the buffer for this LOB</td>
</tr>
<tr>
<td>OCIObjectFreeTemporary()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectGetChunkSize()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectGetLength()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectGetLength2()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectGetStorageLimit()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectIsEqual()</td>
<td>0</td>
</tr>
<tr>
<td>OCIObjectIsOpen()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectIsTemporary()</td>
<td>0</td>
</tr>
<tr>
<td>OCIObjectLoadFromFile()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectLoadFromFile2()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectLocatorAssign()</td>
<td>1 round-trip if either the source or the destination locator refers to a temporary LOB</td>
</tr>
<tr>
<td>OCIObjectLocatorIsInit()</td>
<td>0</td>
</tr>
<tr>
<td>OCIObjectOpen()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectRead()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCIObjectRead2()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCIObjectTrim()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectTrim2()</td>
<td>1</td>
</tr>
<tr>
<td>OCIObjectWrite()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCIObjectWrite2()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCIObjectWriteAppend()</td>
<td>0 or 1</td>
</tr>
<tr>
<td>OCIObjectWriteAppend2()</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
Describe Operation Round-Trips

Table C–4 lists the number of server round-trips required by \texttt{OCIDescribeAny()}, \texttt{OCIAttrGet()}, and \texttt{OCIParamGet()}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Function & Number of Server Round-Trips \\
\hline
\texttt{OCIObjectArrayPin()} & 1 \\
\texttt{OCIObjectCopy()} & 0 \\
\texttt{OCIObjectExists()} & 0 \\
\texttt{OCIObjectFlush()} & 1 \\
\texttt{OCIObjectFree()} & 0 \\
\texttt{OCIObjectGetInd()} & 0 \\
\texttt{OCIObjectGetObjectRef()} & 0 \\
\texttt{OCIObjectGetObjectRef()} & 0 \\
\texttt{OCIObjectGetTypeRef()} & 0 \\
\texttt{OCIObjectIsDirty()} & 0 \\
\texttt{OCIObjectIsLocked()} & 0 \\
\texttt{OCIObjectLock()} & 1 \\
\texttt{OCIObjectMarkDelete()} & 0 \\
\texttt{OCIObjectMarkDeleteByRef()} & 0 \\
\texttt{OCIObjectMarkUpdate()} & 0 \\
\texttt{OCIObjectNew()} & 0 \\
\texttt{OCIObjectPin()} & 1; 0 if the desired object is already in cache \\
\texttt{OCIObjectPinCountReset()} & 0 \\
\texttt{OCIObjectPinTable()} & 1 \\
\texttt{OCIObjectRefresh()} & 1 \\
\texttt{OCIObjectUnmark()} & 0 \\
\texttt{OCIObjectUnmarkByRef()} & 0 \\
\texttt{OCIObjectUnpin()} & 0 \\
\hline
\end{tabular}
\end{table}

Describe Operation Round-Trips

Table C–4 lists the number of server round-trips required by \texttt{OCIDescribeAny()}, \texttt{OCIAttrGet()}, and \texttt{OCIParamGet()}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Function & Number of Server Round-Trips \\
\hline
\texttt{OCIAttrGet()} & 2 round-trips to describe a type if the type objects are not in the object cache

1 round-trip for each collection element, or each type attribute, method, or method argument descriptor. 1 more round-trip if using \texttt{OCI_ATTR_TYPE_NAME}, or \texttt{OCI_ATTR_SCHEMA_NAME} on the collection element, type attribute, or method argument.

0 if all the type objects to be described are already in the object cache following the first \texttt{OCIAttrGet()} call. \\
\texttt{OCIDescribeAny()} & 1 round-trip to get the \texttt{REF} of the type descriptor object \\
\texttt{OCIParamGet()} & 0 \\
\hline
\end{tabular}
\end{table}
Data Type Mapping and Manipulation Function 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>
<thead>
<tr>
<th>Function</th>
<th>Number of Server Round-Trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIColl*()</td>
<td>0; 1 if the collection is not loaded in the cache</td>
</tr>
<tr>
<td>OCIDate*()</td>
<td>0</td>
</tr>
<tr>
<td>OCIIter*()</td>
<td>0; 1 if the collection is not loaded in the cache</td>
</tr>
<tr>
<td>OCINumber*()</td>
<td>0</td>
</tr>
<tr>
<td>OCIRef*()</td>
<td>0</td>
</tr>
<tr>
<td>OCIString*()</td>
<td>0</td>
</tr>
<tr>
<td>OCITable*()</td>
<td>0; 1 if the nested table is not loaded in the cache</td>
</tr>
</tbody>
</table>

Any Type and Data Function 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>
<thead>
<tr>
<th>Function</th>
<th>Number of Server Round-Trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIAnyDataAttrGet()</td>
<td>0; 1 if the type information is not loaded in the cache</td>
</tr>
<tr>
<td>OCIAnyDataAttrSet()</td>
<td>0; 1 if the type information is not loaded in the cache</td>
</tr>
<tr>
<td>OCIAnyDataCollGetElem()</td>
<td>0; 1 if the type information is not loaded in the cache</td>
</tr>
</tbody>
</table>

Other Local Functions

Table C–7 lists the functions that are local and do not require a server round-trip.

<table>
<thead>
<tr>
<th>Local Function Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIAttrGet()</td>
<td>When describing an object type, this call makes one round-trip to fetch the type descriptor object.</td>
</tr>
</tbody>
</table>

OCIAttrSet()
OCIBindArrayOfStruct()
OCIDefineArrayOfStruct()
OCIBindByDynamic()
OCIBindByName()
OCIBindByPos()
OCIBindObject()
<table>
<thead>
<tr>
<th>Local Function Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIDefineDynamic()</td>
<td></td>
</tr>
<tr>
<td>OCIDefineObject()</td>
<td></td>
</tr>
<tr>
<td>OCI_descriptorAlloc()</td>
<td></td>
</tr>
<tr>
<td>OCI_descriptorFree()</td>
<td></td>
</tr>
<tr>
<td>OCIEnvCreate()</td>
<td></td>
</tr>
<tr>
<td>OCIEnvInit()</td>
<td></td>
</tr>
<tr>
<td>OCIErrorGet()</td>
<td></td>
</tr>
<tr>
<td>OCIHandleAlloc()</td>
<td></td>
</tr>
<tr>
<td>OCIHandleFree()</td>
<td></td>
</tr>
<tr>
<td>OCILdaToSvcCtx()</td>
<td></td>
</tr>
<tr>
<td>OCISvcCtxToLda()</td>
<td></td>
</tr>
<tr>
<td>OCIStmtGetBindInfo()</td>
<td></td>
</tr>
<tr>
<td>OCIStmtPrepare()</td>
<td></td>
</tr>
<tr>
<td>OCIStmtRelease()</td>
<td></td>
</tr>
<tr>
<td>OCIStmtPrepare2()</td>
<td></td>
</tr>
<tr>
<td>OCIStmtToLda()</td>
<td></td>
</tr>
</tbody>
</table>
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
- Compiling OCI Applications for Windows
- Linking OCI Applications for Windows
- Running OCI Applications for Windows
- Oracle XA Library
- Using the Object Type Translator for Windows

**What Is Included in the OCI Package for Windows?**

The Oracle Call Interface for Windows package includes:

- Oracle Call Interface (OCI)
- Required Support Files (RSFs)
- Oracle Universal Installer
- Header files for compiling OCI applications
- Library files for linking OCI applications
- Sample programs for demonstrating how to build OCI applications

The OCI for Windows package includes the additional libraries required for linking your OCI programs.

*See Also:* "OCI Instant Client" on page 1-16 for a simplified OCI installation option

**Oracle Directory Structure for Windows**

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

<table>
<thead>
<tr>
<th>Directory Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>\bin</td>
<td>Executable and help files</td>
</tr>
<tr>
<td>\oci</td>
<td>Oracle Call Interface directory for Windows files</td>
</tr>
<tr>
<td>\oci\include</td>
<td>Header files, such as oci.h and ociap.h</td>
</tr>
<tr>
<td>\oci\samples</td>
<td>Sample programs</td>
</tr>
<tr>
<td>\precomp\admin\ottcfg.cfg</td>
<td>Object Type Translator utility and default configuration file</td>
</tr>
</tbody>
</table>

## Sample OCI Programs for Windows

When OCI is installed, a set of sample programs and their corresponding project files are copied to the ORACLE_BASE\ORACLE_HOME\oci\samples subdirectory. Oracle recommends that you build and run these sample programs to verify that OCI has been successfully installed and to familiarize yourself with the steps involved in developing OCI applications.

To build a sample, run a batch file (make.bat) at the command prompt. For example, to build the cdemo1.c sample, enter the following command in the directory samples:

```
C:> make cdemo1
```

After you finish using these sample programs, you can delete them if you choose.

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.

## 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
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.

`oci.lib`

Oracle recommends that applications be linked with `oci.lib`, which takes care of loading the correct versions of the Oracle DLLs.

Client DLL Loading When Using Load Library()

The following directories are searched in this order by 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

Running OCI Applications for Windows

To run an OCI application, ensure that the entire corresponding set of Required Support Files (RSFs) is installed on the computer that is running your OCI application.

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**

<table>
<thead>
<tr>
<th>Component</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>xa.h</td>
<td>ORACLE_BASE\ORACLE_HOME\oci\include</td>
</tr>
</tbody>
</table>

### Compiling and Linking 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**

<table>
<thead>
<tr>
<th>Library</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>oraxa11.lib</td>
<td>ORACLE_BASE\ORACLE_HOME\rdbms\xa</td>
</tr>
<tr>
<td>oci.lib</td>
<td>ORACLE_BASE\ORACLE_HOME\oci\lib\msvc or, ORACLE_BASE\ORACLE_HOME\oci\lib\bc</td>
</tr>
</tbody>
</table>

3. Run `program.exe`.

### Using XA Dynamic Registration

The database supports the use of XA dynamic registration. XA dynamic registration improves the performance of applications 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**
- **Adding a Registry Variable for All Sessions**

#### Adding an Environmental Variable for the Current Session

Adding an environmental variable at the command prompt affects only the current session.

**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.

#### 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.
To Add a Registry Variable:
1. Go to the computer where your TP monitor is installed.
2. Enter the following at the command prompt:
   \C:\> regedt32
   The Registry Editor window appears.
3. Go to HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\HOME.ID.
4. 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.

**XA and TP Monitor Information**

See the following general information about XA and TP monitors:
- Distributed TP: The XA Specification (C193) published by the Open Group
- See the Web site at:
  http://www.opengroup.org/publications/catalog/tp.htm
- Your specific TP monitor documentation:
  - Oracle Database Advanced Application Developer’s Guide, "Developing Applications with Oracle XA," for more information about the Oracle XA Library and using XA dynamic registration
  - For Oracle Tuxedo information see

**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 10g Release 1.
Note that the CASE specification inside the intype files, such as CASE=LOWER, applies only to C identifiers that are not specifically listed, either through a TYPE or TRANSLATE statement in the intype file. It is important to provide the type name with the appropriate cases, such as TYPE Person and Type PeRsOn, in the intype file.

OTT on Windows can be invoked from the command line. 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.
Table E–1 lists the OCI functions that were deprecated in releases previous to Oracle 11g R2 (11.2). In a future release, these functions may become obsolete.

**Table E–1  Deprecated OCI Functions**

<table>
<thead>
<tr>
<th>Function Group</th>
<th>Deprecated Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>OCIEnvInit(), OCIInitialize()</td>
</tr>
<tr>
<td>Statement</td>
<td>OCIStmtFetch()</td>
</tr>
<tr>
<td>Lob</td>
<td>OCILobCopy(), OCILobErase(), OCILobGetLength(), OCILobLoadFromFile(), OCILobRead(), OCILobTrim(), OCILobWrite(), OCILobWriteAppend()</td>
</tr>
<tr>
<td>Streams Advanced Queuing functions</td>
<td>OCIAQListen()</td>
</tr>
</tbody>
</table>
Deprecated Initialize Functions

Table E–2 lists the deprecated Initialize functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIEnvInit()</td>
<td>Initialize an environment handle.</td>
</tr>
<tr>
<td>OCIInitialize()</td>
<td>Initialize OCI process environment.</td>
</tr>
</tbody>
</table>
**OCIEnvInit()**

**Purpose**
Allocates and initializes an OCI environment handle. This function is deprecated.

**Syntax**
```c
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_U CB
  In OCI_DEFAULT mode, the OCI library always mutexes handles.
  The OCI_ENV_NO_U CB mode is used to suppress the calling of the dynamic callback routine OCIEnvCallback() at environment initialization time. The default behavior is to allow such a call to be made.

  **See Also:** "Dynamic Callback Registrations" on page 9-33

- **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().
See Also: "User Memory Allocation" on page 2-12 for more information about the \texttt{xtramemsz} parameter and user memory allocation

Related Functions

\texttt{OCIHandleAlloc()}, \texttt{OCIHandleFree()}, \texttt{OCIEnvCreate()}, \texttt{OCITerminate()}
OCIInitialize()

Purpose

Initializes the OCI process environment. This function is deprecated.

Syntax

```c
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_THREAD - 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_THREAD, 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_THREAD, 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.

cctxp (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" on page 8-24
  for information about using the OCI to write multithreaded
  applications
- Chapter 11 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 (*)(void *)) 0, (void (*)(void *)) 0, (void (*)(void *)) 0);

Related Functions

OCIHandleAlloc(), OCIHandleFree(), OCIEnvCreate(), OCITerminate()
Deprecated Statement Functions

Table E–3 lists the deprecated Statement functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIStmtFetch()</td>
<td>Fetch rows from a query.</td>
</tr>
</tbody>
</table>
OCIStmtFetch()

Purpose

Fetches rows from a query. This function is deprecated. Use OCIStmtFetch2().

Syntax

```c
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 Functions

OCIStmtExecute(), OCIStmtFetch2()
## Deprecated Lob Functions

Table E–4 lists the deprecated LOB functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCILobCopy()</td>
<td>Copy all or part of one LOB to another.</td>
</tr>
<tr>
<td>OCILobErase()</td>
<td>Erase a portion of a LOB.</td>
</tr>
<tr>
<td>OCILobGetLength()</td>
<td>Get length of a LOB.</td>
</tr>
<tr>
<td>OCILobLoadFromFile()</td>
<td>Load a LOB from a BFILE.</td>
</tr>
<tr>
<td>OCILobRead()</td>
<td>Read a portion of a LOB.</td>
</tr>
<tr>
<td>OCILobTrim()</td>
<td>Truncate a LOB.</td>
</tr>
<tr>
<td>OCILobWrite()</td>
<td>Write into a LOB.</td>
</tr>
<tr>
<td>OCILobWriteAppend()</td>
<td>Write data beginning at the end of a LOB.</td>
</tr>
</tbody>
</table>
OCILobCopy()

Purpose

Copies all or a portion of a LOB value into another LOB value. This function is deprecated. Use OCILobCopy2().

Syntax

```c
sword OCILobCopy ( OCISvcCtx *svchp,
  OCIError *errhp,
  OCILobLocator *dst_locp,
  OCILobLocator *src_locp,
  ub4 amount,
  ub4 dst_offset,
  ub4 src_offset );
```

Parameters

See: "OCILobCopy2()" on page 17-37
**OCILobErase()**

**Purpose**

Erases a specified portion of the internal LOB data starting at a specified offset. This function is deprecated. Use **OCILobErase2()**.

**Syntax**

```c
sword OCILobErase ( OCISvcCtx *svchp,
                   OCIError *errhp,
                   OCILobLocator *locp,
                   ub4 *amount,
                   ub4 offset );
```

**Parameters**

See: "**OCILobErase2()**" on page 17-43
OCILobGetLength()

Purpose

Gets the length of a LOB. This function is deprecated. Use OCILobGetLength2().

Syntax

```c
sword OCILobGetLength ( OCISvcCtx *svchp,
                         OCIError *errhp,
                         OCILobLocator *locp,
                         ub4 *lenp );
```

Parameters

See: "OCILobGetLength2()" on page 17-58
**OCILOBLoadFromFile()**

**Purpose**

Loads and copies all or a portion of the file into an internal LOB. This function is deprecated. Use **OCILOBLoadFromFile2()**.

**Syntax**

```c
sword OCILOBLoadFromFile ( OCISvcCtx *svchp,
OCIError *errhp,
OCILOBLocator *dst_locp,
OCILOBLocator *src_locp,
ub4 amount,
ub4 dst_offset,
ub4 src_offset );
```

**Parameters**

See: "OCILOBLoadFromFile2()" on page 17-66
OCILobRead()

Purpose

Reads a portion of a LOB or BFILE, as specified by the call, into a buffer. This function is deprecated. Use OCILobRead2().

Syntax

```c
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>
<thead>
<tr>
<th>LOB or BFILE</th>
<th>Input</th>
<th>Output with Fixed-Width Client-Side Character Set</th>
<th>Output with Varying-Width Client-Side Character Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOBs and BFILES</td>
<td>bytes</td>
<td>bytes</td>
<td>bytes</td>
</tr>
<tr>
<td>CLOBs and NCLOBs</td>
<td>characters</td>
<td>characters</td>
<td>bytes ¹</td>
</tr>
</tbody>
</table>

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 (CLOBs, NCLOBs) it is the number of characters from the beginning of the LOB, for binary LOBs or BFILES it is the number of bytes. The first position is 1.

If you use streaming (by polling or a callback), 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 CLOBs and for NCLOBs (`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.

csf rm (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 BFILES, 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 CLOBs and NCLOBs:
- 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 CLOBs and NCLOBs:
- When reading a CLOB or NCLOB value, look at the amtp parameter after every call to OCILobRead() to see how much of the buffer is filled. When the return value is in characters (as when the client-side character set is fixed-width), then convert this
value to bytes and determine how much of the buffer is filled. When 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.

**See Also:**
- "PL/SQL REF CURSORS and Nested Tables in OCI" on page 5-32 for additional information about Unicode format
- Oracle Database SecureFiles and Large Objects Developer's Guide for a description of `BFILE`
- The demonstration programs included with your Oracle Database installation for a code sample showing the use of LOB reads and writes.
- Appendix B, "OCI Demonstration Programs"
- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for general information about piecewise OCI operations
- "Polling Mode Operations in OCI" on page 2-27

**Related Functions**

`OCIErrorGet()`, `OCILobRead2()`, `OCILobWrite()`, `OCILobWrite2()`, `OCILobFileSetName()`, `OCILobWriteAppend()`, `OCILobWriteAppend2()`
OCILobTrim()  

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  
See: "OCILobTrim2()" on page 17-80
OCILobWrite()

Purpose

Writes a buffer into a LOB. This function is deprecated. Use OCILobWrite2().

Syntax

```c
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()

<table>
<thead>
<tr>
<th>LOB or BFILE</th>
<th>Input with Fixed-Width [Client-Side Character Set]</th>
<th>Input with Varying-Width [Client-Side Character Set]</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOBs and BFILEs</td>
<td>bytes</td>
<td>bytes</td>
<td>bytes</td>
</tr>
<tr>
<td>CLOBs and NCLOBs</td>
<td>characters</td>
<td>bytes¹</td>
<td>characters</td>
</tr>
</tbody>
</table>

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 CLOBs and NCLOBs 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.

csfm (IN)
The character set form of the buffer data. The csfm parameter must be consistent with the type of the LOB.

The csfm 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 (csfm) 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. BFILES are not allowed, because they are read-only. If the LOB is a BLOB, the csid and csfm parameters are ignored.

If the client-side character set is varying-width, then the input amount is in bytes and the output amount is in characters for CLOBs and NCLOBs. The input amount refers to the number of bytes of data that the user wants to write into the LOB and not the number of bytes in the bufp, which is specified by buflen. 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.

**See Also:**

- "PL/SQL REF CURSORs and Nested Tables in OCI" on page 5-32 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.
- Appendix B, "OCI Demonstration Programs"
- "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-33 for general information about piecewise OCI operations
- "Polling Mode Operations in OCI" on page 2-27

**Related Functions**

OCIErrorGet(), OCILobRead(), OCILobRead2(), OCILobAppend(), OCILobCopy(), OCILobCopy2(), OCILobWriteAppend(), OCILobWriteAppend2(), OCILobWrite2()
OCILobWriteAppend()

Purpose

Writes data starting at the end of a LOB. This function is deprecated. Use OCILobWriteAppend2().

Syntax

```c
sword OCILobWriteAppend ( OCISvcCtx *svchp,
OCIError *errhp,
OCILobLocator *locp,
um4 *amtp,
void  *bufp,
um4 buflen,
um1 piece,
void  *ctxp,
OCICallbackLobWrite   (cbfp)
) (void
    void     *ctxp,
    void     *bufp,
um4     *lenp,
um1     *piecep
) (void
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()

<table>
<thead>
<tr>
<th>LOB or BFILE</th>
<th>Input with Fixed-Width Client-Side Character Set</th>
<th>Input with Varying-Width Client-Side Character Set</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOBs and BFILES</td>
<td>bytes</td>
<td>bytes</td>
<td>bytes</td>
</tr>
<tr>
<td>CLOBs and NCLOBs</td>
<td>characters</td>
<td>bytes 1</td>
<td>characters</td>
</tr>
</tbody>
</table>

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 CLOBs or NCLOBs.

**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`.

---

**OCILOBWriteAppend()**

Oracle Call Interface Programmer’s Guide
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 CLOBs or NCLOBs.

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.

See Also: “Improving LOB Read/Write Performance” on page 7-8

Related Functions

OCIErrorGet(), OCILobRead(), OCILobRead2(), OCILobAppend(), OCILobCopy(), OCILobCopy2(), OCILobWrite(), OCILobWrite2(), OCILobWriteAppend2()
Deprecated Streams Advanced Queuing Functions

Table E–8 lists the deprecated Streams Advanced Queuing functions that are described in this section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIAQListen()</td>
<td>Listen on one or more queues on behalf of a list of agents.</td>
</tr>
</tbody>
</table>
OCIAQListen()

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,
ub4 wait,
OCIAQAgent **agent,
ub4 flags);

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.

agent_list (IN)
List of agents for which to monitor messages.

num_agents (IN)
Number of agents in the agent list.

wait (IN)
Timeout interval for the listen call.

agent (OUT)
Agent for which there is a message. OCIAgent is an OCI descriptor.

flags (IN)
Not currently used; pass as OCI_DEFAULT.

Comments
This is a blocking call that returns when there is a message ready for consumption for an agent in the list. If there are no messages found when the wait time expires, an error is returned.

Related Functions
OCIAQEnq(), OCIAQDeq(), OCIAQListen2(), OCISvcCtxToLda(),
OCISubscriptionEnable(), OCISubscriptionPost(), OCISubscriptionRegister(),
OCISubscriptionUnRegister()
Symbols

, A-8

Numerics

64-bit integer support, 3-9

A
administration handle, 10-17
  attributes, A-22
    description, 10-17
ADR, 10-32
ADR base location, 10-32
ADR, controlling creation, 10-35
ADRCI command-line tool, 10-33
ADT
  See object type
advanced queuing
  agent descriptor, 2-9
  dequeue options descriptor, 2-9
  descriptor, 2-12
  enqueue options descriptor, 2-9
  listen options descriptor, 2-9
  message properties descriptor, 2-9
  notification descriptor, 2-9
advantages
  OCI, 1-2
allocation duration
  example, 14-12
  of objects, 14-11
ANSI DATE descriptor, 2-9
AnyData, 12-20
AnyDataSet, 12-20
AnyType, 12-20
application context, 8-20
application initialization, 2-14
AQ
  See Oracle Streams Advanced Queuing,
argument attributes, 6-12
arrays
  binds, 12-26
  defines, 12-28
  DML, maximum rows, 5-4
  skip parameter for, 5-19
  arrays of structures, 5-18
  indicator variables, 5-19
  OCI calls used, 5-19
  skip parameters, 5-19
attribute descriptor object, 12-20
attributes
  administration handle, A-22
  authentication information handle, A-14
  bind handle, A-34
  complex object retrieval (COR) descriptor, A-40
  complex object retrieval (COR) handle, A-40
  connection pool handle, A-22
  continuous query notification, A-57
  continuous query notification descriptor, A-58
  define handle, A-37
    description, A-37
  describe handle, A-39
direct path loading
  column array handle, A-69
  column parameter, A-71
  context handle, A-63
  function context handle, A-68
  handle, A-62
  stream handle, A-70
environment handle, A-2
error handle, A-8
event handle, A-77
LOB locator, A-39
notification descriptor, A-60
OCIAQAgent descriptor, A-50
OCIAQDeqOptions descriptor, A-43
OCIAQEnqOptions descriptor, A-41
OCIAQMsgProperties descriptor, A-46
OCIServerDNs descriptor, A-51
of handles, 2-8
of objects, 11-12
of parameter descriptors, 6-4
of parameters, 6-4
parameter descriptor, A-39
process handle, A-75
server handle, A-11
service context handle, A-8
session pool handle, A-24
statement handle, A-27
subscription handle, A-51
transaction handle, A-26
user session handle, A-14
authentication
  by Distinguished Name, 8-11
  by X.509 Certificate, 8-12
management, 8-7
authentication information handle, 2-4
authorize functions, 16-3
Automatic Diagnostic Repository (ADR), 10-32
Automatic Workload Repository (AWR), 10-30

B

BASICFILE parameter, 7-22
batch error mode, 4-7
batch jobs, authenticating users in, 8-9
BFILE
data type, 3-18
maximum size, 7-4
BFILE data type locator descriptor, 2-9
bin directory, D-2
BINARY_DOUBLE data type, 3-5
BINARY_FLOAT data type, 3-5
bind and define in statement caching, 9-28
bind functions, 16-62
bind handle
  attributes, A-34
  description, 2-6
bind operations, 4-4, 5-1, 12-25
  associations made, 5-1
  example, 5-5
  initializing variables, 5-2
  LOBs, 5-8
  named data types, 12-25
  named versus positional, 5-2
  OCI array interface, 5-3
  OCI_DATA_AT_EXEC mode, 5-13
  PL/SQL, 5-4
  positional versus named, 5-2
  REF CURSOR variables, 5-13
  REFs, 12-25
  steps used, 5-5
bind placeholders, 4-5
binding
  arrays, 12-26
  buffer expansion, 5-29
  multiple buffers, 5-20
  OCI_NUMBER, 12-29
  PL/SQL placeholders, 2-29
  summary, 5-7
BLOB data type, 3-18
blocking modes, 2-27
branches
  detaching, 8-4
  preparing multiple, 8-6
  resuming, 8-4
buffer expansion during binding, 5-29
buffered messaging, 9-52
buffering LOB operations, 7-9
building OCI applications, B-1

C

C data types
  manipulating with OCI, 12-3
  cache functions
    server round-trips, C-4
  cache, client result, 10-23
  callbacks
    dynamic registrations, 9-33
    for LOB operations, 7-11
    for reading LOBs, 7-11
    for writing LOBs, 7-13
    from external procedures, 9-37
    LOB streaming interface, 7-11
    parameter modes, 16-82
    registration for transparent application failover, 9-40
    restrictions, 9-35
    user-defined functions, 9-29
  canceling a cursor, 17-7
  canceling OCI calls, 2-25
  cartridge functions, 20-1
  CASE OTT parameter, 15-24
  change notification descriptor, 2-9
  CHAR
    external data type, 3-14
    character set conversion of Unicode, 2-35
    character set form, 5-26
    character set ID, 5-26
    Unicode, A-35, A-37
    character-length semantics, 5-29, 5-30, 6-17
  CHARZ
    external data type, 3-15
    checkerr() listing, 17-166
    CHUNK size, 7-4
    client result cache, 10-23
    maximum size, 10-29
    CLIENTCONTEXT namespace, 8-20
  CLOB
data type, 3-18
  code
    example programs, B-1
    list of demonstration programs, B-1
  CODE OTT parameter, 15-22
  coding guidelines
    reserved words, 2-26
  coherency
    of object cache, 14-3
  collections
    attributes, 6-10
    data manipulation functions, 12-14
    describing, 6-1
    description, 12-14
    functions for manipulating, 12-14
    multilevel, 12-17
    scanning functions, 12-15
  column objects
    direct path loading of, 13-15
  columns
    attributes, 6-3, 6-11
    commit, 2-19
in object applications, 14-11
one-phase for global transactions, 8-5
two-phase for global transactions, 8-5
compatibility
of releases in OCI, 1-12
compiling
OCI application, D-2
OCI with Oracle XA, D-4
XA Library, D-3
complex object descriptor, 2-9
complex object retrieval, 11-15
descriptor attributes, A-40
handle, 2-7
handle attributes, A-40
implementing, 11-17
navigational prefetching, 11-18
complex object retrieval (COR) descriptor, 2-12
COMPRESS, 7-22
CONFIG OTT parameter, 15-23
configuration files, D-2
location, D-2
connect functions, 16-3
connection, 2-14
connection mode
nonblocking, 2-27
connection pool handle
attributes, A-22
description, 2-8
initializing, 9-4
connection pooling, 9-1, 9-23
code example, 9-7
creating, 9-4
database resident, 9-13
connection pools and TAF, 9-3
consistency
of object cache, 14-3
continuous query notification, 10-1
descriptor attributes, A-57
descriptor example, 10-7
copying
objects, 11-23
COR
See complex object retrieval
creating
objects, 11-23
cursor cancelation, 17-7

D
data cartridges
OCI functions, 2-1, 20-1
data definition language
SQL statements, 1-5
data manipulation language
SQL statements, 1-5
data structures, 2-3
See also descriptors
See also handles
data types
ANSI DATE, 3-19
BFILE, 3-18
binding and defining, 12-28
BLOBs (binary large objects), 3-18
CLOB, 3-18
conversions, 3-21
direct path loading, 13-3, A-73
direct path loading, 13-3, A-73
external, 3-3, 3-6
FILE, 3-18
for piecewise operations, 5-34
internal, 3-3
internal codes, 3-3
INTERVAL DAY TO SECOND, 3-20
INTERVAL YEAR TO MONTH, 3-20
manipulating with OCI, 12-3
mapping and manipulation functions, C-6
mapping from Oracle to C, 12-2
mapping, Oracle methodology, 12-3
mapping, OTT utility, 15-8
NCLOB, 3-19
Oracle, 3-1
TIMESTAMP, 3-19
TIMESTAMP WITH LOCAL TIME ZONE, 3-19
TIMESTAMP WITH TIME ZONE, 3-19
database connection
for object applications, 11-7
database resident connection pooling, 9-13
database type attributes
type OCI_PTYPE_DATABASE, 6-15
databases
attributes, 6-15
creating, 6-1
shutting down, 10-17
starting up, 10-17
DATE
date cache, 13-13
date descriptor, 2-11
DATE, ANSI
data type, 3-19
datetime
avoiding unexpected results, 3-20
datetime and date
migration rules, 3-24
datetime descriptor, 2-11
DDL
See data definition language
DEDUPPLICATE, 7-22
default file name extensions
OTT utility, 15-30
default name mapping
OTT utility, 15-30
define
arrays, 12-28
return and error codes, 2-21
define functions, 16-62
define handle
attributes, A-37
description, 2-6
define operations, 4-12, 5-13, 12-26
example, 5-14
LOBs, 5-16
named data types, 12-26
piecewise fetch, 5-17
PL/SQL output variables, 5-17
REFs, 12-27
steps used, 5-14
defining
multiple buffers, 5-20
OCI Number, 12-29
deletes
positioned, 2-25
demonstration files location, 2-1
demonstration programs, B-1, D-2
list, B-1
describe
explicit, 4-9, 4-11
explicit and implicit, 6-3
implicit, 4-9, 4-10
of collections, 6-1
of databases, 6-1
of packages, 6-1
of schemas, 6-1
of sequences, 6-1
of stored functions, 6-1
of stored procedures, 6-1
of synonyms, 6-1
of tables, 6-1
of types, 6-1
of views, 6-1
select list, 4-9
describe functions, 16-62
describe handle
attributes, A-39
description, 2-7
describe operation
server round-trips, C-5
describing the stored procedure example, 6-20
descriptor, 2-9
advanced queuing
agent, 2-9
dequeue options, 2-9
enqueue options, 2-9
listen options, 2-9
message properties, 2-9
notification, 2-9
allocating, 2-14
ANSI DATE, 2-9
BFILE data type locator, 2-9
change notification, 2-9
complex object, 2-9
complex object retrieval, 2-12
distinguished names of the database servers in a registration request, 2-9
functions, 16-47
INTERVAL DAY TO SECOND, 2-9
INTERVAL YEAR TO MONTH, 2-9
LOB data type locator, 2-9
objects, 12-20
parameter, 2-11
read-only parameter, 2-9
result set, 2-9
row change, 2-9
ROWID, 2-9, 2-11
snapshot, 2-9, 2-10
table change, 2-9
TIMESTAMP, 2-9
TIMESTAMP WITH LOCAL TIME ZONE, 2-9
TIMESTAMP WITH TIME ZONE, 2-9
user callback, 2-9
detaching branches, 8-4
DIAG_ADR_ENABLED sqlnet.ora parameter, 10-35
DIAG_DDE_ENABLED sqlnet.ora parameter, 10-35
DIAG_RESTRITED, 10-35
DIAG_SIGHANDLER_ENABLED, 10-35
direct path loading, 13-1
attribute
OCI_ATTR_DIRPATH_NO_INDEX_ ERRORS, 13-30
column array handle attributes, A-69
column parameter attributes, A-71
data types of columns, 13-3, A-73
dynamic registration
Oracle XA Library, D-4
dynamically linked applications, 1-12
DML
See data manipulation language
DRCP (database resident connection pooling), 9-13
duration
example, 14-12
of objects, 14-11
dynamic registration
Oracle XA Library, D-4
dynamically linked applications, 1-12

E

edition-based redefinition, 8-22
editions, 8-22
embedded objects
fetching, 11-11
embedded SQL, 1-7
mixing with OCI calls, 1-7
empty LOB, inserting, 7-15
ENCRYPT, 7-22
enhanced DML array
feature, 4-7
environment handle
attributes, A-2
description, 2-5
error checking example, 17-166
error codes
define calls, 2-21
 navigational functions, 18-4
error handle
attributes, A-8
description, 2-5
errors
 handling, 2-20
 handling in object applications, 11-26
ERRTYPE OTT parameter, 15-23
evaluation context
attributes, 6-16
evaluation context attributes, 6-16
event callback, 9-45
event handle attributes, A-77
demonstration programs, B-1
executing SQL statements, 4-5
execution
against multiple servers, 4-4
modes, 4-6
snapshots, 4-6
explicit describe, 4-9, 6-18
extensions
OTT utility default file name, 15-30
external data types, 3-3, 3-6
CHAR, 3-14
CHARZ, 3-15
conversions, 3-21
DATE, 3-13
FLOAT, 3-11
INTEGER, 3-11
LOB, 3-17
LONG, 3-12
LONG RAW, 3-14
LONG VARCHAR, 3-14
LONG VARRAW, 3-14
named data types, 3-16
NUMBER, 3-8
RAW, 3-13
REF, 3-16
ROWID, 3-16
SQLT_BLOB, 3-17
SQLT_CLOB, 3-17
SQLT_NCLOB, 3-17
SQLT_NTY, 3-16
SQLT_REFT, 3-16
STRING, 3-11
UNSIGNED, 3-14
VARCHAR, 3-12
VARCHAR2, 3-7
VARNUM, 3-12
VARRAW, 3-14
external procedure functions
return codes, 20-1
with_context type, 20-2
external procedures
OCI callbacks, 9-37
externally initialized context, 8-15

F
failover
callback example, 9-40
failover callbacks, 9-37
structure and parameters, 9-39
Fast Application Notification (FAN), 9-13
fault diagnosability, 10-32
read or write client driver layer name, A-18
fetch
piecewise, 5-33, 5-38
fetch operation, 4-13
LOB data, 4-13
setting prefetch count, 4-13
FILE
associating with operating system file, 7-2
data type, 3-18
FLOAT
external data type, 3-11
flushing, 14-8
object changes, 11-10
objects, 14-8
freeing
 objects, 11-23, 14-7
functions
attributes, 6-6

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-3
global transactions, 8-2
globalization support, 2-30
OCI functions, 2-1
GTRID. See transaction identifier

H
HA event notification, 9-43
handle attributes, 2-8
handle functions, 16-47
handle type constants, 2-3
handles, 2-3
administration handle, 10-17, A-22
advantages of, 2-5
allocating, 2-4, 2-14
bind handle, 2-6, A-34
C data types, 2-3
child freed when parent freed, 2-5
complex object retrieval handle, 2-7, A-40
collection pool handle, 2-8, 9-4, A-22
define handle, 2-6, A-37
describe handle, 2-7, A-39
direct path loading, 2-7
environment handle, 2-5, A-2
error handle, 2-5, A-8
freeing, 2-4
process attributes, A-75
server handle, 2-5, A-11
service context handle, 2-5
statement handle, 2-6, A-27
subscription, 2-7, 9-56
transaction handle, 2-6, A-26
types, 2-3
user session handle, 2-5, A-14

header files
location of, 1-11, 2-1, D-2
oratypes.h, 3-28

HFILE OTT parameter, 15-23

implicit describe, 4-9, 6-18
include directory, D-2
indicator variables, 2-24
arrays of structures, 5-19
for named data types, 2-23, 2-24
for REFs, 2-23, 2-24
named data type defines, 12-27
PL/SQL OUT binds, 12-27
REF defines, 12-27
with named data type bind, 12-26
with REF bind, 12-26
INITFILE OTT parameter, 15-22
INITFUNC OTT parameter, 15-23
initialize
all buffers, 5-4
functions, 16-3
init.ora security parameters, 8-23
inserting an empty LOB, 7-15
inserts
piecewise, 5-33, 5-35
Instant Client, 1-16
Instant Client Light (English), 1-24
INTEGER external data type, 3-11
internal codes for data types, 3-3
internal data types, 3-3
conversions, 3-21
INTERVAL DAY TO SECOND data type, 3-20
INTERVAL DAY TO SECOND descriptor, 2-9
interval descriptor, 2-11
INTERVAL YEAR TO MONTH data type, 3-20
INTERVAL YEAR TO MONTH descriptor, 2-9
intype file
providing when running OTT utility, 15-6
structure of, 15-25
INTTYPE File Assistant, D-5
INTTYPE OTT parameter, 15-22
IP address
IPv4, A-53
IPv6, A-53

K
KEEP_DUPLICATES, 7-22
key words, 2-26

L
LDAP registration of publish-subscribe notification, 9-59
libraries
oci.lib, D-3
linking
OCI application, D-3
OCI with Oracle XA, D-4
XA Library, D-3
list attributes
type OCI_PTYPE_LIST, 6-14
lists
attributes, 6-14
lmsgen utility, 2-36
LoadLibrary, D-3
LOB and LONG bind restrictions, 5-9
LOB data type locator descriptor, 2-9
LOB functions, 17-17
server round-trips, C-3
LOB locator, 2-10
attributes, A-39
LOBs
amount and offset parameters, 17-18
attributes of transient objects, 7-3
binding, 5-8
buffering, 7-9
callbacks, 7-11
character sets, 17-18
creating, 7-2
creating temporary, 7-15
defining, 5-16
duration of temporary, 7-15
temporary does not work, 9-42
fetching data, 4-13
fixed-width character sets, 17-18
freeing temporary, 7-15
greater than 4 GB, 7-4
locators, 2-10
modifying, 7-2
OCI functions, 7-8
prefetching, 7-19
size maximum, 7-4
temporary, 7-14
locators, 2-9
for LOB data type, 2-10
locking, 14-10
objects, 14-10
optimistic model, 14-11
LONG
external data type, 3-12
LONG RAW
external data type, 3-14
LONG VARCHAR
external data type, 3-14
LONG VARRAW
external data type, 3-14

M
make.bat file, D-2
makefiles, 2-1, 2-2, B-1
marking objects, 14-7
meta-attributes
of objects, 11-12
of persistent objects, 11-12
of transient objects, 11-14
method descriptor object, 12-20
method results
type OCI_PTYPE_TYPE_RESULT, 6-12
migration
7.x to 8.0, 1-15
session, 8-8, 16-31
miscellaneous functions, 17-162
multiple servers
executing statement against, 4-4
multithreaded development
basic concepts, 8-24
multithreading, 8-24
mutexes, 8-24

N
named data types
binding, 12-25
binding and defining, 12-28
defining, 12-26
definition, 3-16
external data types, 3-16
indicator variables, 2-24
indicator variables for, 2-23
name-value pair attributes
type OCI_PTYPE_NAME_VALUE, 6-17
native double, 3-20
native float, 3-20
navigation, 14-14
navigational functions
error codes, 18-4
return values, 18-3
terminology, 18-3
NCHAR
issues, 5-26
NCLOB
data type, 3-19
nested table
direct path loading of, 13-14
element ordering, 12-17
functions for manipulating, 12-16
new release, relinking, 1-12
NLS_LANG, 2-30
NLS_NCHAR, 2-30
NOCOMPRESS, 7-22
NOENCRYPT, 7-22
nonblocking mode, 2-27
non-deferred linkage no longer supported, 8-24
nonfinal object tables
direct path loading of, 13-26
notification descriptor attributes, A-60
NULL indicator
setting for an object attribute, 11-23
NULL indicator struct, 11-21
generated by OTT, 11-6
nullness
of objects, 11-21
NULLS
atomic, 11-21
inserting, 2-24
inserting into database, 2-23
inserting using indicator variables, 2-23
NUMBER
external data type, 3-8
O
object applications
commit, 14-11
database connection, 11-7
rollback, 14-11
object cache, 14-1
coherency, 14-3
consistency, 14-3
initializing, 11-6
loading objects, 14-5
memory parameters, 14-4
operations on, 14-4
removing objects, 14-5
setting the size of, 14-4
object functions
server round-trips, C-4
object identifiers, 11-25
for persistent objects, 11-3
object references, 11-25
See REFs
object run-time environment
initializing, 11-6
object tables
direct path loading of, 13-25
object type
representing in C applications, 11-5
Object Type Translator utility
See OTT utility
object view, 11-14
objects
accessing with OCI, 15-17
allocation duration, 14-11
array pin, 11-9
attributes, 11-12
manipulating, 11-9
client-side cache, 14-1
copying, 11-23
creating, 11-23
duration, 14-11
flushing, 14-8
flushing changes, 11-10
freeing, 11-23, 14-7
lifetime, 18-1
LOB attributes of transient objects, 7-3
locking, 14-10
manipulating with OCI, 15-17
marking, 11-10, 14-7
memory layout of instance, 14-13
memory management, 14-1
meta-attributes, 11-12
navigation, 14-14
simple, 14-14
NCHAR and NVARCHAR2 attribute of, 12-2
NULL values, 11-21
OCI object application structure, 11-2
persistent, 11-3
pin count, 11-21
pin duration, 14-11
pinning, 11-8, 14-5
refreshing, 14-9
secondary memory, 14-13
terminology, 11-3, 18-1
top-level memory, 14-13
transient, 11-3, 11-4
types, 11-3, 18-1
unmarking, 14-8
unpinning, 14-7
use with OCI, 11-1
OCI
aborting calls, 2-25
accessing and manipulating objects, 15-17
advantages, 1-2
object support, 1-3
Oracle XA Library, D-4
overview, 1-1
parts of, 1-3
sample programs, D-2
OCI applications
compiling, 1-2, D-2
general structure, 2-2
initialization example, 2-18
linking, 1-2, D-3
running, D-3
steps, 2-12
structure, 2-2
structure using objects, 11-2
terminating, 2-20
using the OTT utility with, 15-16
with objects
initializing, 11-6
OCI directory, D-2
OCI environment
initializing for objects, 11-6
OCI functions
canceling calls, 2-25
data cartridges, 2-1
globalization, 2-1
not supported, 1-14
obsolete, 1-13
return codes, 2-20, 2-22
OCI handle types, 2-3
OCI navigational functions, 14-15
flush functions, 14-16
mark functions, 14-16
meta-attribute accessor functions, 14-16
miscellaneous functions, 14-16
naming scheme, 14-15
pin/unpin/free functions, 14-15
OCI process
initializing for objects, 11-6
OCI program
See OCI applications
OCI relational functions
connect, authorize, and initialize, 16-3
guide to reference entries, 20-1
Oracle Streams Advanced Queuing and publish-subscribe, 17-88, E-26
OCI_ATTR_ACCESS_BANNER, 8-24
server handle attribute, A-11
OCI_ATTR_ACTION, 8-17, 16-35
user session handle attribute, A-14
OCI_ATTR_ADMIN_PFILE
administration handle attribute, A-22
OCI_ATTR_AGENT_ADDRESS
OCIAQAgent descriptor attribute, A-50
OCI_ATTR_AGENT_NAME
OCIAQAgent descriptor attribute, A-50
OCI_ATTR_AGENT_PROTOCOL
OCIAQAgent descriptor attribute, A-51
OCI_ATTR_ALLOC_DURATION
environment handle attribute, A-3
OCI_ATTR_APPCTX_ATTR, 8-16
user session handle attribute, A-14
OCI_ATTR_APPCTX_LIST, 8-16
user session handle attribute, A-15
OCI_ATTR_APPCTX_NAME, 8-16
user session handle attribute, A-15
OCI_ATTR_APPCTX_SIZE, 8-16, 16-35
user session handle attribute, A-15
OCI_ATTR_APPCTX_VALUE, 8-16
user session handle attribute, A-16
OCI_ATTR_AQ_NTFN_GROUPING_MSGID_ARRAY
notification descriptor attribute, A-60
OCI_ATTR_AQ_NTFN_GROUPING_COUNT, 9-62
notification descriptor attribute, A-60
OCI_ATTR_AQ_NTFN_GROUPING_MSGID_ARRAY, 9-62
OCI_ATTR_ATTEMPTS
OCIAQMsgProperties descriptor attribute, A-46
OCI_ATTR_AQNTFN_GROUPING, 9-62
OCI_ATTR_AQNTFN_GROUPING_COUNT, 9-62
OCI_ATTR_AQNTFN_GROUPING_MSGID_ARRAY, 9-62
OCI_ATTR_BREAK_ON_NET_TIMEOUT
server handle attribute, A-11
OCI_ATTR_BUF_ADDR
  direct path loading stream handle attribute, A-70
OCI_ATTR_BUF_SIZE
  direct path loading context handle attribute, A-63
  stream handle attribute, A-70
OCI_ATTR_CACHE
  attribute, 6-11
OCI_ATTR_CACHE_ARRAYFLUSH, 14-9
  environment handle attribute, A-3
OCI_ATTR_CACHE_MAX_SIZE, 14-4
  environment handle attribute, A-3
OCI_ATTR_CACHE_OPT_SIZE, 14-4
  environment handle attribute, A-4
OCI_ATTR_CALL_TIME, 8-17
  user session handle attribute, A-16
OCI_ATTR_CATALOG_LOCATION
  attribute, 6-15
OCI_ATTR_CERTIFICATE, 8-12, 16-35
  user session handle attribute, A-16
OCI_ATTR_CHAR_COUNT
  bind handle attribute, A-34
  define handle attribute, A-37
OCI_ATTR_CHAR_SIZE, 6-12
  attribute, A-17, 6-18
OCI_ATTR_CHAR_USED, 6-12
  attribute, A-17, 6-18
OCI_ATTR_CHARSET_FORM, 5-26
  attribute, A-9, 6-10, 6-12, 6-14
  bind handle attribute, A-35
  define handle attribute, A-37
OCI_ATTR_CHARSET_ID, 5-26
  attribute, A-9, 6-10, 6-12, 6-14, 6-15
  bind handle attribute, A-35
  define handle attribute, A-37
  direct path loading column parameter attribute, A-72
  context handle attribute, A-63
OCI_ATTR_CHDES_DBNAME, 10-6
  continuous query notification descriptor attribute, A-58
OCI_ATTR_CHDES_NFTYPE, 10-6
  continuous query notification descriptor attribute, A-59
OCI_ATTR_CHDES_QUERIES, 10-6
OCI_ATTR_CHDES_ROW_OPFLAGS, 10-7
  continuous query notification descriptor attribute, A-59
OCI_ATTR_CHDES_ROW_ROWID, 10-7
  continuous query notification descriptor attribute, A-59
OCI_ATTR_CHDES_TABLE_CHANGES, 10-6
  continuous query notification descriptor attribute, A-59
OCI_ATTR_CHDES_TABLE_NAME, 10-6
  continuous query notification descriptor attribute, A-60
OCI_ATTR_CHDES_TABLE_OVFFLAGS, 10-6
  continuous query notification descriptor attribute, A-60
OCI_ATTR_CHNF_CHANGELAG, 10-5
  continuous query notification attribute, A-58
OCI_ATTR_CHNF_OPERATIONS, 10-5
  continuous query notification attribute, A-58
OCI_ATTR_CHNF_REGHANDLE
  statement handle attribute, A-27
OCI_ATTR_CHNF_TABLENAMES, 10-4
  continuous query notification attribute, A-58
OCI_ATTR_CLIENT_IDENTIFIER, 8-12, 16-35
  user session handle attribute, A-17
OCI_ATTR_CLIENT_INFO, 8-17, 16-35
  user session handle attribute, A-17
OCI_ATTR_CLUSTERED
  attribute, A-6-6
OCI_ATTR_COL_COUNT
  direct path loading column array handle attribute, A-69
OCI_ATTR_CONN_BUSY_COUNT
  connection pool handle attribute, A-23
OCI_ATTR_CONN_INCR
  connection pool handle attribute, A-24
OCI_ATTR_CONN_MAX
  connection pool handle attribute, A-23
OCI_ATTR_CONN_MIN
  connection pool handle attribute, A-23
OCI_ATTR_CONN_NOWAIT
  connection pool handle attribute, A-23
OCI_ATTR_CONN_OPEN_COUNT
  connection pool handle attribute, A-23
OCI_ATTR_CONN_TIMEOUT, 9-4
  connection pool handle attribute, A-22
OCI_ATTR_CONNECTION_CLASS, 16-35
  user session handle attribute, A-17
OCI_ATTR_CONSUMER_NAME, 9-62, A-43
notification descriptor attribute, A-61
OCI_ATTR_CORRELATION
OCIAQDeqOptions descriptor attribute, A-43
OCIAQMsgProperties descriptor attribute, A-47
OCI_ATTR_CQ_QUERYID, 10-5
statement handle attribute, A-28
OCI_ATTR_CQDES_OPERATION, 10-6
direct path loading handle attribute, A-62
OCI_ATTR_CQDES_QUERYID, 10-6
direct path loading handle attribute, A-62
OCI_ATTR_CQDES_TABLE_CHANGES, 10-6
direct path loading handle attribute, A-62
OCI_ATTR_CURRENT_POSITION
attribute, 4-15
statement handle attribute, A-28
OCI_ATTR_CURRENT_SCHEMA, 16-35
user session handle attribute, A-18
OCI_ATTR_CURSOR_COMMIT_BEHAVIOR
attribute, 6-15
OCI_ATTR_DATA_SIZE, 6-12, 6-17
column attribute, 6-8, 6-10, 6-12, 6-13, 6-18
direct path loading column parameter attribute, A-73
OCI_ATTR_DATA_TYPE
direct path loading column parameter attribute, A-73
OCI_ATTR_DATEFORMAT
direct path loading column parameter attribute, A-74
context handle attribute, A-63
OCI_ATTR_DBDOMAIN
event handle attribute, A-77
OCI_ATTR_DBNAME
event handle attribute, A-77
OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE, 7-19, 16-35
user session handle attribute, A-18
OCI_ATTR_DELAY
OCIAQMsgProperties descriptor attribute, A-47
OCI_ATTR_DEQ_MODE
OCIAQDeqOptions descriptor attribute, A-43
OCI_ATTR_DEQ_MSGID
OCIAQDeqOptions descriptor attribute, A-44
OCI_ATTR_DEQCOND
OCIAQDeqOptions descriptor attribute, A-44
OCI_ATTR_DESC_PUBLIC, 16-88
OCI_ATTR_DIRPATH_DCACHE_DISABLE, 13-14
direct path loading context handle attribute, A-63
OCI_ATTR_DIRPATH_DCACHE_HITS, 13-14
direct path loading context handle attribute, A-64
OCI_ATTR_DIRPATH_DCACHE_MISSES, 13-13
direct path loading context handle attribute, A-64
OCI_ATTR_DIRPATH_DCACHE_NUM, 13-13
direct path loading context handle attribute, A-64
OCI_ATTR_DIRPATH_EXPR_TYPE
direct path function context handle attribute, A-68
function context attribute, 13-30
OCI_ATTR_DIRPATH_INDEX_MAINT_METHOD
direct path loading context handle attribute, A-64
OCI_ATTR_DIRPATH_MODE
direct path loading context handle attribute, A-65
OCI_ATTR_DIRPATH_NO_INDEX_ERRORS, 13-30
direct path loading context handle attribute, A-65
OCI_ATTR_DIRPATH_NOLOG
direct path loading context handle attribute, A-65
OCI_ATTR_DIRPATH_OBJ_CONSTR, 13-28
direct path context attribute, 13-28
direct path loading context handle attribute, A-65
OCI_ATTR_DIRPATH_OID
direct path loading column parameter attribute, A-74
OCI_ATTR_DIRPATH_PARALLEL, 13-2
direct path loading context handle attribute, A-66
OCI_ATTR_DIRPATH_SID
column array attribute, 13-34
direct path loading column parameter attribute, A-74
OCI_ATTR_DIRPATH_SKIPINDEX_METHOD
direct path loading context handle attribute, A-66
OCI_ATTR_DISTINGUISHED_NAME, 8-11, 8-12, 16-35
user session handle attribute, A-18
OCI_ATTR_DML_ROW_OFFSET
error handle attribute, A-8
OCI_ATTR_DN_COUNT
OCIServerDNs descriptor attribute, A-51
OCI_ATTR_DRIVER_NAME, 16-35
user session handle attribute, A-18
OCI_ATTR_DURATION
attribute, 6-6
OCI_ATTR_EDITION, 8-22, 16-35
user session handle attribute, A-19
OCI_ATTR_ENCAPSULATION
attribute, 6-9
OCI_ATTR_ENQ_TIME
OCIAQMsgProperties descriptor attribute, A-47
OCI_ATTR_ENV
server handle attribute, A-11
service context handle attribute, A-9
statement handle attribute, A-28
OCI_ATTR_ENV_CHARSET_ID, 2-31
environment handle attribute, A-4
OCI_ATTR_ENV_NCHARSET_ID, 2-31
environment handle attribute, A-4
OCI_ATTR_ENV_UTF16
environment handle attribute, A-4
OCI_ATTR_EVALCONTEXT_NAME
attribute, 6-16
OCI_ATTR_EVALCONTEXT_OWNER
attribute, 6-16
OCI_ATTR_EVALUATION_FUNCTION
attribute, 6-16
OCI_ATTR_EVENTTYPE
event handle attribute, A-77
OCI_ATTR_EVTCBK, 9-45
environment handle attribute, A-5

Index-10
OCI_ATTR_EVTCTX, 9-45
environment handle attribute, A-5
OCI_ATTR_EXCEPTION_QUEUE
OCIAQMsgProperties descriptor attribute, A-47
OCI_ATTR_EXPIRATION
OCIAQMsgProperties descriptor attribute, A-48
OCI_ATTR_EXTERNAL_NAME, 8-5
server handle attribute, A-11
OCI_ATTR_FETCH_ROWID, 10-21
statement handle attribute, A-28
OCI_ATTR_FOCBK
server handle attribute, A-12
OCI_ATTR_FSPRECISION
attribute, 6-9
OCI_ATTR_HA_SOURCE
event handle attribute, A-77
OCI_ATTR_HA_SRVFIRST, 9-45
event handle attribute, A-78
OCI_ATTR_HA_SRVNEXT, 9-45
event handle attribute, A-78
OCI_ATTR_HA_STATUS
event handle attribute, A-78
OCI_ATTR_HA_TIMESTAMP
event handle attribute, A-78
OCI_ATTR_HAS_DEFAULT
attribute, 6-13
OCI_ATTR_HAS_FILE
attribute, 6-7
OCI_ATTR_HAS_LOB
attribute, 6-7
OCI_ATTR_HAS_NESTED_TABLE
attribute, 6-7
OCI_ATTR_HEAPALLOC
environment handle attribute, A-5
OCI_ATTR_HOSTNAME
event handle attribute, A-79
OCI_ATTR_HW_MARK
attribute, 6-11
OCI_ATTR_IN_V8_MODE
server handle attribute, A-12
service context handle attribute, A-9
OCI_ATTR_INCR
attribute, 6-11
OCI_ATTR_INDEXONLY
attribute, 6-6
OCI_ATTR_INITIAL_CLIENT_ROLES, 8-12, 16-35
user session handle attribute, A-19
OCI_ATTR_INSTNAME
event handle attribute, A-79
OCI_ATTR_INSTSTARTTIME
event handle attribute, A-79
OCI_ATTR_INTERNAL_NAME, 8-5
server handle attribute, A-12
OCI_ATTR_JOMODE
attribute, 6-13
OCI_ATTR_IS_CONSTRUCTOR
attribute, 6-9
OCI_ATTR_IS_DESTRUCTOR
attribute, 6-9
OCI_ATTR_IS_FINAL_METHOD
attribute, 6-9
OCI_ATTR_IS_FINAL_TYPE
attribute, 6-8
OCI_ATTR_IS_INCOMPLETE_TYPE
attribute, 6-7
OCI_ATTR_IS_INSTANTIABLE_METHOD
attribute, 6-9
OCI_ATTR_IS_INSTANTIABLE_TYPE
attribute, 6-8
OCI_ATTR_IS_INVOKER_RIGHTS
attribute, 6-6, 6-7, 6-8
OCI_ATTR_IS_MAP
attribute, 6-9
OCI_ATTR_IS_NULL
attribute, 6-12, 6-13
OCI_ATTR_IS_OPERATOR
attribute, 6-9
OCI_ATTR_IS_ORDER
attribute, 6-9
OCI_ATTR_IS_OVERRIDEING_METHOD
attribute, 6-9
OCI_ATTR_IS_PREDEFINED_TYPE
attribute, 6-7
OCI_ATTR_IS_RNDS
attribute, 6-9
OCI_ATTR_IS_RNPS
attribute, 6-9
OCI_ATTR_IS_SELFISH
attribute, 6-9
OCI_ATTR_IS_SUBTYPE
attribute, 6-8
OCI_ATTR_IS_SYSTEM_GENERATED_TYPE
attribute, 6-7
OCI_ATTR_IS_SYSTEM_TYPE
attribute, 6-7
OCI_ATTR_IS_TEMPORARY
attribute, 6-6
OCI_ATTR_IS_TRANSIENT_TYPE
attribute, 6-7
OCI_ATTR_IS_TYPED
attribute, 6-6
OCI_ATTR_IS_WNDS
attribute, 6-9
OCI_ATTR_IS_WNPS
attribute, 6-9
OCI_ATTR_LDAP_AUTH, 9-59
environment handle attribute, A-5
OCI_ATTR_LDAP_CRED, 9-59
environment handle attribute, A-6
OCI_ATTR_LDAP_CTX, 9-59
environment handle attribute, A-6
OCI_ATTR_LDAP_HOST, 9-59
environment handle attribute, A-6
OCI_ATTR_LDAP_PORT, 9-59
environment handle attribute, A-6
OCI_ATTR_LEVEL
attribute, 6-13
OCI_ATTR_LFPRECISION
attribute, 6-9
OCI_ATTR_LINK
attribute, 6-11, 6-14
OCI_ATTR_LIST_ACTION_CONTEXT
attribute, 6-16
OCI_ATTR_LIST_ARGUMENTS
attribute, 6-4, 6-6, 6-9, 6-13
OCI_ATTR_LIST_COLUMNS
attribute, 6-5
    direct path loading
        context handle attribute, A-66
        function context handle attribute, A-68
OCI_ATTR_LIST_OBJECTS
attribute, 6-15
OCI_ATTR_LIST_RULES
attribute, 6-16
OCI_ATTR_LIST_SCHEMAS
attribute, 6-15
OCI_ATTR_LIST_SUBPROGRAMS
attribute, 6-7
OCI_ATTR_LIST_TABLE_ALIASES
attribute, 6-16
OCI_ATTR_LIST_TYPE
attribute, 6-7
OCI_ATTR_LIST_TYPE_ATTRS
attribute, 6-7
OCI_ATTR_LIST_TYPE_METHODS
attribute, 6-7
OCI_ATTR_LIST_VARIABLE_TYPES
attribute, 6-16
OCI_ATTR_LOBEMPTY
LOB locator attribute, A-40
OCI_ATTR_LOBPREFETCH_LENGTH, 7-20
define handle attribute, A-38
OCI_ATTR_LOBPREFETCH_SIZE, 7-20
define handle attribute, A-38
OCI_ATTR_LOCKING_MODE
attribute, 6-15
OCI_ATTR_MAP_METHOD
attribute, 6-7
OCI_ATTR_MAX
attribute, 6-11
OCI_ATTR_MAX_CATALOG_NAMELEN
attribute, 6-15
OCI_ATTR_MAX_COLUMN_NAMELEN
attribute, 6-15
OCI_ATTR_MAX_PROC_LEN
attribute, 6-15
OCI_ATTR_MAXCHAR_SIZE
attribute, 5-28, 5-29
    bind handle attribute, A-35
    define handle attribute, A-38
OCI_ATTR_MAXDATA_SIZE
attribute, 5-28
    bind handle attribute, A-35
    use with binding, 5-28
OCI_ATTR_MEMPOOL_APPNAME
process handle attribute, A-75
OCI_ATTR_MEMPOOL_HOMENAME
process handle attribute, A-76
OCI_ATTR_MEMPOOL_INSTNAME
process handle attribute, A-76
OCI_ATTR_MEMPOOL_SIZE
    process handle attribute, A-76
OCI_ATTR_MIGSESSION
    user session handle attribute, A-19
OCI_ATTR_MIN
attribute, 6-11
OCI_ATTR_MODULE, 8-17, 16-35
    user session handle attribute, A-20
OCI_ATTR_MSG_DELIVERY_MODE, 17-89, 17-91
    OCIAQDeqOptions descriptor attribute, A-44
    OCIAQEnqOptions descriptor attribute, A-41
    OCIAQMsgProperties descriptor attribute, A-48
OCI_ATTR_MSG_PROP, 9-62
    notification descriptor attribute, A-61
OCI_ATTR_MSG_STATE
    OCIAQMsgProperties descriptor attribute, A-48
OCI_ATTR_NAME
attribute, 6-6, 6-8, 6-9, 6-10, 6-11, 6-12, 6-13, 6-17
column array attribute, 13-32
direct path loading
column parameter attribute, A-74
column attribute, A-66
function context attribute, 13-29
function context handle attribute, A-68
OCI_ATTR_NAVIGATION
    OCIAQDeqOptions descriptor attribute, A-44
OCI_ATTR_NCHARSET_ID
attribute, 6-15
OCI_ATTR_NFY_FLAGS, 9-62
    notification descriptor attribute, A-61
OCI_ATTR_NFY_MSGID, 9-62
    notification descriptor attribute, A-61
OCI_ATTR_NONBLOCKING_MODE
    server handle attribute, 2-28, A-12
OCI_ATTR_NOWAIT_SUPPORT
attribute, 6-15
OCI_ATTR_NUM_COLS
attribute, 6-5
direct path loading
column array attribute, A-69
function context attribute, 13-30
function context handle attribute, A-68
direct path loading context handle attribute, A-67
OCI_ATTR_NUM_DML_ERRORS
    statement handle attribute, A-29
OCI_ATTR_NUM_ELEMS
attribute, 6-10
OCI_ATTR_NUM_HANDLES
attribute, 6-14
OCI_ATTR_NUM_ROWS
attribute, 13-34
direct path loading
column array handle attribute, A-69
function context attribute, A-67
function context handle attribute, A-69
function context attribute, 13-31
OCI_ATTR_NUM_TYPE_ATTRS
attribute, 6-7
OCI_ATTR_NUM_TYPE_METHODS
attribute, 6-7
OCI_ATTR_OBJ_ID
OCI_ATTR_OBJ_NAME
attribute, 6-4

OCI_ATTR_OBJ_SCHEMA
attribute, 6-5

OCI_ATTR_OBJECT
environment handle attribute, A-6

OCI_ATTR_OBJECT_DETECTCHANGE
environment handle attribute, 14-11, A-7

OCI_ATTR_OBJECT_NEWNOTNULL
environment handle attribute, A-7

OCI_ATTR_OBJID
attribute, 6-5, 6-11

OCI_ATTR_ORDER
attribute, 6-11

OCI_ATTR_ORDER_METHOD
attribute, 6-8

OCI_ATTRORIGINAL_MSGID
OCLAQMsgProperties descriptor attribute, A-49

OCI_ATTROVERLOAD_ID
attribute, 6-6

OCI_ATTRPARAM
describe handle attribute, A-39
use when an attribute is itself a descriptor, 16-51

OCI_ATTRPARAM_COUNT
describe handle attribute, A-39
statement handle attribute, A-29

OCI_ATTRPARSE_ERROR_OFFSET statement
handle attribute, A-30

OCI_ATTRPARTITIONED
attribute, 6-6

OCI_ATTR_PASSWORD, 8-13, 16-35
user session handle attribute, A-20

OCI_ATTRPDPRC
bind handle attribute, A-36
define handle attribute, A-38

OCI_ATTRPDSCL
bind handle attribute, A-36
define handle attribute, A-39

OCI_ATTRPIN_DURATION
environment handle attribute, A-7

OCI_ATTRPINOPTION
environment handle attribute, A-7

OCI_ATTRPOSITION
attribute, 6-13

OCI_ATTRPRECISION
attribute, 6-3, 6-8, 6-10, 6-12, 6-13
direct path loading column parameter
attribute, A-74

OCI_ATTRPREFETCH_MEMORY
statement handle attribute, A-30

OCI_ATTRPREFETCH_ROWS
statement handle attribute, A-30

OCI_ATTR_PRIORITY
OCLAQMsgProperties descriptor attribute, A-49

OCI_ATTRPROC_MODE
process handle attribute, A-76

OCI_ATTR_PROXY_CLIENT, 2-17
user session handle attribute, A-20

OCI_ATTR_PROXY_CREDENTIALS, 8-11, 16-35
user session handle attribute, A-20

OCI_ATTRPTYPER
attribute, 6-5
possible values of, 6-5

OCI_ATTRPURITY, 16-35
user session handle attribute, A-21

OCI_ATTRPURITY_DEFAULT
user session handle attribute, A-21

OCI_ATTRPURITY_NEW, 9-16

OCI_ATTRPURITY_SELF, 9-16

OCI_ATTRQUEUE_NAME, 9-62
notification descriptor attribute, A-61

OCI_ATTRRADIX
attribute, 6-13

OCI_ATTRRDBA
attribute, 6-6

OCI_ATTRRECIPIENT_LIST
OCLAQMsgProperties descriptor attribute, A-49

OCI_ATTRREF_TDO
attribute, 6-5, 6-7, 6-9, 6-10, 6-12, 6-14

OCI_ATTRRELATIVE_MSGID
OCLAQEnqOptions descriptor attribute, A-41

OCI_ATTRRELATIVE_MSGID enqueue
option, 9-52

OCI_ATTRROW_COUNT, 4-15
direct path loading
column array handle attribute, A-70
stream handle attribute, A-70
statement handle attribute, A-30

OCI_ATTRROWID
statement handle attribute, A-31

OCI_ATTRROWS_FETCHED, 4-15
statement handle attribute, A-31

OCI_ATTRROWS_RETURNED
bind handle attribute, A-36
use with callbacks, 5-26

OCI_ATTRSAVEPOINT_SUPPORT
attribute, 6-15

OCI_ATTRSCALE
attribute, 6-8, 6-10, 6-12, 6-13
direct path loading column parameter
attribute, A-75

OCIATTR_SCHEMA_NAME
attribute, 6-8, 6-10, 6-11, 6-12, 6-13
direct path loading context handle attribute, A-67

OCI_ATTRSERVER_ID
OCLAQMsgProperties descriptor attribute, A-50

OCI_ATTRSEQ
attributes, 6-11

OCI_ATTRSEQUENCE enqueue option, 9-52

OCI_ATTRSEQUENCE_DEVIATION
OCLAQEnqOptions descriptor attribute, A-42

OCIATTRSERVER
service context handle attribute, A-9

OCI_ATTRSERVER_DN
OCIServerDNs descriptor attribute, A-51

OCI_ATTRSERVER_DNS
subscription handle attribute, A-51

OCI_ATTRSERVER_GROUP, 8-10
server handle attribute, A-13
OCI_LOCK_NONE, 14-10
OCI_LOCK_X, 14-10
OCI_LOCK_X_NOWAIT, 14-10
    parameter usage, 14-10
OCI_LOGON2_STMTCACHE, 9-27
OCI_LTYPE_ARG_FUNC list attribute, 6-14
OCI_LTYPE_ARG_PROC list attribute, 6-14
OCI_LTYPE_COLUMN list attribute, 6-14
OCI_LTYPE_DB_SCH list attribute, 6-14
OCI_LTYPE_SCH_OBJ list attribute, 6-14
OCI_LTYPE_SUBPRG list attribute, 6-14
OCI_LTYPE_TYPE_ARG_FUNC list attribute, 6-14
OCI_LTYPE_TYPE_ARG_PROC list attribute, 6-14
OCI_LTYPE_TYPE_ATTR list attribute, 6-14
OCI_LTYPE_TYPE_METHOD list attribute, 6-14
OCI_MAJOR_VERSION, client library
    version, 17-164
OCI_MIGRATE, 8-8
    OCISessionBegin() mode, 16-30
OCI_MINOR_VERSION, client library
    version, 17-164
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_ON, 16-14,
    16-18
OCI_NCHAR_LITERAL_REPLACE_OFF, 16-14,
OCI CollGetElemArray(), 19-10
OCI CollIsLocator(), 19-12
OCI CollMax(), 19-13
OCI CollSize(), 19-14
OCI CollTrim(), 19-16
OCI ComplexObject
use of, 11-17
OCI ComplexObject Comp
use of, 11-17
OCI ConnectionPoolCreate(), 16-7
OCI ConnectionPoolDestroy(), 16-9
OCI ContextClearValue(), 20-15
OCI ContextGenerateKey(), 20-16
OCI ContextGetValue(), 20-17
OCI ContextSetValue(), 20-18
OCI Date, 12-5
binding and defining, 12-5, 12-28
manipulation usage example, 12-5
OCI Date AddDays(), 19-27
OCI Date AddMonths(), 19-28
OCI Date Assign(), 19-29
OCI Date Check(), 19-30
OCI Date Compare(), 19-32
OCI Date DaysBetween(), 19-33
OCI Date From Text(), 19-34
OCI Date Get Date(), 19-36
OCI Date Get Time(), 19-37
OCI Date Last Day(), 19-38
OCI Date Next Day(), 19-39
OCI Date Set Date(), 19-40
OCI Date Set Time(), 19-41
OCI Date Sys Date(), 19-42
OCI Date Time Assign(), 19-43
OCI Date Time Check(), 19-44
OCI Date Time Compare(), 19-46
OCI Date Time Construct(), 19-47
OCI Date Time Convert(), 19-49
OCI Date Time From Array(), 19-50
OCI Date Time From Text(), 19-51
OCI Date Time Get Date(), 19-53
OCI Date Time Get Time(), 19-54
OCI Date Time Get Time Zone Name(), 19-55
OCI Date Time Get Time Zone Offset(), 19-56
OCI Date Time Interval Add(), 19-57
OCI Date Time Interval Sub(), 19-58
OCI Date Time Subtract(), 19-59
OCI Date Time Sys Time Stamp(), 19-60
OCI Date Time To Array(), 19-61
OCI Date Time To Text(), 19-62
OCI Date To Text(), 19-64
OCI Date Zone To Zone(), 19-66
OCI DB Shutdown(), 16-10
OCI DB Startup(), 16-12
OCI Define Array Of Struct(), 16-77
OCI Define By Pos(), 16-78
OCI Define Dynamic(), 16-82
OCI Define Object(), 16-84
OCI Describe Any(), 16-86
limitations, 6-2
overview, 6-1

usage examples, 6-18
OCI Descriptor Alloc(), 16-54
OCI Descriptor Freet(), 16-56
OCI Dir Path Abort(), 17-107
OCI Dir Path Col Array context, 13-4
OCI Dir Path Col Array Entry Get(), 17-108
OCI Dir Path Col Array Entry Set(), 17-109
OCI Dir Path Col Array Reset(), 17-111
OCI Dir Path Col Array Row Get(), 17-112
OCI Dir Path Col Array To Stream(), 17-113
OCI Dir Path Ctx context, 13-4
OCI Dir Path Data Save(), 17-115
OCI Dir Path Finish(), 17-116
OCI Dir Path Flush Row(), 17-117
OCI Dir Path Load Stream(), 17-118
OCI Dir Path Prepare(), 17-119
OCI Dir Path Stream context, 13-4
OCI Dir Path Stream Reset(), 17-120
OCI Duration
use of, 14-6, 14-11
OCI Duration Begin(), 17-20, 20-9
OCI Duration End(), 17-21, 20-10
OCI Env Create(), 16-13
OCI Env Init(), E-3
OCI Env Nls Create(), 2-30, 5-27, 16-17
OCI Error Get(), 17-165
OCI Event handle, 9-44
OCI Event Callback data type, 9-45
OCI Ext Proc Alloc Call Memory(), 20-4
OCI Ext Proc Get Env(), 20-5
OCI Ext Proc Raise Excp(), 20-6
OCI Ext Proc Raise Excp With Msg(), 20-7
OCI Extract From File(), 20-20
OCI Extract From List(), 20-21
OCI Extract From Str(), 20-22
OCI Extract Init(), 20-23
OCI Extract Reset(), 20-24
OCI Extract Set Key(), 20-25
OCI Extract Set Num Keys(), 20-27
OCI Extract Term(), 20-28
OCI Extract To Bool(), 20-29
OCI Extract To Int(), 20-30
OCI Extract To List(), 20-31
OCI Extract To OCINum(), 20-32
OCI Extract To Str(), 20-33
OCI File Close(), 20-35
OCI File Exists(), 20-36
OCI File Flush(), 20-37
OCI File Get Length(), 20-38
OCI File Init(), 20-39
OCI File Object data structure, 20-34
OCI File Open(), 20-40
OCI File Read(), 20-42
OCI File Seek(), 20-43
OCI File Term(), 20-44
OCI File Write(), 20-45
OCI Format Init(), 20-47
OCI Format String(), 20-48
OCI Format Term(), 20-53
OCI Handle Alloc(), 16-57

Index-18
OCIHandleFree(), 16-58
OCIind
  use of, 11-22
OCIInitialize(), E-5
OCIIntervalAdd(), 19-68
OCIIntervalAssign(), 19-69
OCIIntervalCheck(), 19-70
OCIIntervalCompare(), 19-72
OCIIntervalDivide(), 19-73
OCIIntervalFromNumber(), 19-74
OCIIntervalFromText(), 19-75
OCIIntervalGetDaySecond(), 19-77
OCIIntervalGetYearMonth(), 19-78
OCIIntervalMultiply(), 19-79
OCIIntervalSetDaySecond(), 19-80
OCIIntervalSetYearMonth(), 19-81
OCIIntervalSubtract(), 19-82
OCIIntervalToNumber(), 19-83
OCIIntervalToText(), 19-84
OCIIOV struct, 5-20
OCIIter
  binding and defining, 12-14
  usage example, 12-15
OCIIterCreate(), 19-17
OCIIterDelete(), 19-18
OCIIterGetCurrent(), 19-19
OCIIterInit(), 19-20
OCIIterNext(), 19-21
OCIIterPrev(), 19-23
OCILdaToSvcCtx(), 17-168
oci.lib, D-3
OCILobAppend(), 17-22
OCILobArrayRead(), 7-20, 17-24
OCILobArrayWrite(), 17-28
OCILobAssign(), 17-32
OCILobCharSetForm(), 17-34
OCILobCharSetId(), 17-35
OCILobClose(), 17-36
OCILobCopy(), E-10
OCILobCopy2(), 17-37
OCILobCreateTemporary(), 17-39
OCILobDisableBuffering(), 17-41
OCILobEnableBuffering(), 17-42
OCILobErase(), E-11
OCILobErase2(), 17-43
OCILobFileClose(), 17-45
OCILobFileCloseAll(), 17-46
OCILobFileExists(), 17-47
OCILobGetFileName(), 17-48
OCILobFileIsOpen(), 17-50
OCILobFileOpen(), 17-51
OCILobFileName(), 17-52
OCILobFlushBuffer(), 17-53
OCILobFreeTemporary(), 17-54
OCILobGetChunkSize(), 7-5, 17-55
OCILobGetContentType(), 17-56
OCILobGetLength(), E-12
OCILobGetLength2(), 17-58
OCILobGetOptions(), 17-59
OCILobGetStorageLimit(), 17-61
OCILobIsEqual(), 17-62
OCILobIsOpen(), 17-63
OCILobIsTemporary(), 17-65
OCILobLoadFromFile(), E-13
OCILobLoadFromFile2(), 17-66
OCILobLocatorAssign(), 17-68
OCILobLocatorIsInit(), 17-70
OCILobOpen(), 17-71
OCILobRead(), E-14
OCILobRead2(), 17-73
OCILobSetContentType(), 17-77
OCILobSetOptions(), 17-79
OCILobTrim(), E-18
OCILobTrim2(), 17-80
OCILobWrite(), E-19
OCILobWrite2(), 17-81
OCILobWriteAppend(), E-23
OCILobWriteAppend2(), 17-85
OCILockOpt
  possible values, 18-24, 18-50
OCILogoff(), 16-21
OCILogon(), 16-22
  using, 2-14
OCILogon2(), 16-24
OCIMemoryAlloc(), 20-11
OCIMemoryFree(), 20-12
OCIMemoryResize(), 20-13
OCIMessageClose(), 22-64
OCIMessageGet(), 22-65
OCIMessageOpen(), 22-66
OCIMultiByteInSizeToWideChar(), 22-16
OCIMultiByteStrCaseConversion(), 22-17
OCIMultiByteStrcat(), 22-18
OCIMultiByteStrcmp(), 22-19
OCIMultiByteStrcpy(), 22-20
OCIMultiByteStrlen(), 22-21
OCIMultiByteStrncmp(), 22-22
OCIMultiByteStrncpy(), 22-23
OCIMultiByteStrncpy(), 22-25
OCIMultiByteStrnDisplayLength(), 22-26
OCIMultiByteToWideChar(), 22-27
OCINlsCharSetConvert(), 22-60
OCINlsCharSetIdToName(), 22-4
OCINlsCharsetNameTold(), 22-5
OCINlsEnvironmentVariableGet(), 5-27, 22-6
OCINlsGetInfo(), 2-31, 22-8
OCINlsNameMap(), 22-13
OCINlsNumericInfoGet(), 22-11
OCINumber, 12-9
  bind example, 12-29
  binding and defining, 12-9, 12-28
  define example, 12-29
  usage examples, 12-10
OCINumberAbs(), 19-88
OCINumberAdd(), 19-89
OCINumberArcCos(), 19-90
OCINumberArcSin(), 19-91
OCINumberArcTan(), 19-92
OCINumberArcTan2(), 19-93
<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCINumberAssign()</td>
<td>19-94</td>
</tr>
<tr>
<td>OCINumberCeil()</td>
<td>19-95</td>
</tr>
<tr>
<td>OCINumberCmp()</td>
<td>19-96</td>
</tr>
<tr>
<td>OCINumberCos()</td>
<td>19-97</td>
</tr>
<tr>
<td>OCINumberDec()</td>
<td>19-98</td>
</tr>
<tr>
<td>OCINumberDiv()</td>
<td>19-99</td>
</tr>
<tr>
<td>OCINumberExp()</td>
<td>19-100</td>
</tr>
<tr>
<td>OCINumberFloor()</td>
<td>19-101</td>
</tr>
<tr>
<td>OCINumberFromInt()</td>
<td>19-102</td>
</tr>
<tr>
<td>OCINumberFromReal()</td>
<td>19-103</td>
</tr>
<tr>
<td>OCINumberFromText()</td>
<td>19-104</td>
</tr>
<tr>
<td>OCINumberHypCos()</td>
<td>19-106</td>
</tr>
<tr>
<td>OCINumberHypSin()</td>
<td>19-107</td>
</tr>
<tr>
<td>OCINumberHypTan()</td>
<td>19-108</td>
</tr>
<tr>
<td>OCINumberInc()</td>
<td>19-109</td>
</tr>
<tr>
<td>OCINumberIntPower()</td>
<td>19-110</td>
</tr>
<tr>
<td>OCINumberIsInt()</td>
<td>19-111</td>
</tr>
<tr>
<td>OCINumberIsZero()</td>
<td>19-112</td>
</tr>
<tr>
<td>OCINumberLn()</td>
<td>19-113</td>
</tr>
<tr>
<td>OCINumberLog()</td>
<td>19-114</td>
</tr>
<tr>
<td>OCINumberMod()</td>
<td>19-115</td>
</tr>
<tr>
<td>OCINumberMul()</td>
<td>19-116</td>
</tr>
<tr>
<td>OCINumberNeg()</td>
<td>19-117</td>
</tr>
<tr>
<td>OCINumberPower()</td>
<td>19-118</td>
</tr>
<tr>
<td>OCINumberPrec()</td>
<td>19-119</td>
</tr>
<tr>
<td>OCINumberRound()</td>
<td>19-120</td>
</tr>
<tr>
<td>OCINumberSetPi()</td>
<td>19-121</td>
</tr>
<tr>
<td>OCINumberSetZero()</td>
<td>19-122</td>
</tr>
<tr>
<td>OCINumberShift()</td>
<td>19-123</td>
</tr>
<tr>
<td>OCINumberSign()</td>
<td>19-124</td>
</tr>
<tr>
<td>OCINumberSin()</td>
<td>19-125</td>
</tr>
<tr>
<td>OCINumberSqr()</td>
<td>19-126</td>
</tr>
<tr>
<td>OCINumberSub()</td>
<td>19-127</td>
</tr>
<tr>
<td>OCINumberTan()</td>
<td>19-128</td>
</tr>
<tr>
<td>OCINumberToInt()</td>
<td>19-129</td>
</tr>
<tr>
<td>OCINumberToReal()</td>
<td>19-130</td>
</tr>
<tr>
<td>OCINumberToRealArray()</td>
<td>19-131</td>
</tr>
<tr>
<td>OCINumberToText()</td>
<td>19-132</td>
</tr>
<tr>
<td>OCINumberTrunc()</td>
<td>19-134</td>
</tr>
<tr>
<td>OCIObjectArrayPin()</td>
<td>18-46</td>
</tr>
<tr>
<td>OCIObjectCopy()</td>
<td>18-29</td>
</tr>
<tr>
<td>OCIObjectExists()</td>
<td>18-22</td>
</tr>
<tr>
<td>OCIObjectFlush()</td>
<td>18-11</td>
</tr>
<tr>
<td>OCIObjectFree()</td>
<td>18-48</td>
</tr>
<tr>
<td>OCIObjectGetAttr()</td>
<td>18-31</td>
</tr>
<tr>
<td>OCIObjectGetInd()</td>
<td>18-33</td>
</tr>
<tr>
<td>OCIObjectGetObjectRef()</td>
<td>18-34</td>
</tr>
<tr>
<td>OCIObjectGetProperty()</td>
<td>18-23</td>
</tr>
<tr>
<td>OCIObjectGetTypeRef()</td>
<td>18-35</td>
</tr>
<tr>
<td>OCIObjectIsDirty()</td>
<td>18-26</td>
</tr>
<tr>
<td>OCIObjectIsLocked()</td>
<td>18-27</td>
</tr>
<tr>
<td>OCIObjectLifetime</td>
<td></td>
</tr>
<tr>
<td>possible values, 18-24</td>
<td></td>
</tr>
<tr>
<td>OCIObjectLock()</td>
<td>18-36</td>
</tr>
<tr>
<td>OCIObjectLockNoWait()</td>
<td>18-37</td>
</tr>
<tr>
<td>OCIObjectMarkDelete()</td>
<td>18-16</td>
</tr>
<tr>
<td>OCIObjectMarkDeleteByRef()</td>
<td>18-17</td>
</tr>
<tr>
<td>OCIObjectMarkStatus</td>
<td></td>
</tr>
<tr>
<td>possible values, 18-25</td>
<td></td>
</tr>
<tr>
<td>OCIObjectMarkUpdate()</td>
<td>18-18</td>
</tr>
<tr>
<td>OCIObjectNew()</td>
<td>18-38</td>
</tr>
<tr>
<td>OCIObjectPin()</td>
<td>18-50</td>
</tr>
<tr>
<td>OCIObjectPinCountReset()</td>
<td>18-52</td>
</tr>
<tr>
<td>OCIObjectPinTable()</td>
<td>18-53</td>
</tr>
<tr>
<td>OCIObjectRefresh()</td>
<td>18-12</td>
</tr>
<tr>
<td>OCIObjectSetAttr()</td>
<td>18-41</td>
</tr>
<tr>
<td>OCIObjectUnmark()</td>
<td>18-19</td>
</tr>
<tr>
<td>OCIObjectUnmarkByRef()</td>
<td>18-20</td>
</tr>
<tr>
<td>OCIObjectUnpin()</td>
<td>18-55</td>
</tr>
<tr>
<td>OCIParmGet()</td>
<td>16-59</td>
</tr>
<tr>
<td>used for describing</td>
<td>4-10</td>
</tr>
<tr>
<td>OCIParmSet()</td>
<td>16-61</td>
</tr>
<tr>
<td>OCIPasswordChange()</td>
<td>17-169</td>
</tr>
<tr>
<td>OCIPing()</td>
<td>17-171</td>
</tr>
<tr>
<td>OCIPinOpt</td>
<td></td>
</tr>
<tr>
<td>use of</td>
<td>14-5</td>
</tr>
<tr>
<td>OCIRef</td>
<td>12-18</td>
</tr>
<tr>
<td>binding and defining</td>
<td>12-13</td>
</tr>
<tr>
<td>manipulation usage</td>
<td>12-13</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>OCIRaw</td>
<td>12-13</td>
</tr>
<tr>
<td>AllocSize()</td>
<td>19-136</td>
</tr>
<tr>
<td>OCIRawAssignBytes()</td>
<td>19-137</td>
</tr>
<tr>
<td>OCIRawAssignRaw()</td>
<td>19-138</td>
</tr>
<tr>
<td>OCIRawPtr()</td>
<td>19-139</td>
</tr>
<tr>
<td>OCIRawResize()</td>
<td>19-140</td>
</tr>
<tr>
<td>OCIRawSize()</td>
<td>19-141</td>
</tr>
<tr>
<td>OCIRaw</td>
<td></td>
</tr>
<tr>
<td>binding and defining</td>
<td>12-18</td>
</tr>
<tr>
<td>usage example</td>
<td>12-19</td>
</tr>
<tr>
<td>OCIRawAssign()</td>
<td>19-143</td>
</tr>
<tr>
<td>OCIRawClear()</td>
<td>19-144</td>
</tr>
<tr>
<td>OCIRawFromHex()</td>
<td>19-145</td>
</tr>
<tr>
<td>OCIRawHexSize()</td>
<td>19-146</td>
</tr>
<tr>
<td>OCIRawIsNull()</td>
<td>19-147</td>
</tr>
<tr>
<td>OCIRawIsEqual()</td>
<td>19-148</td>
</tr>
<tr>
<td>OCIRawIsNull()</td>
<td>19-149</td>
</tr>
<tr>
<td>OCIRawToHex()</td>
<td>19-149</td>
</tr>
<tr>
<td>OCIRawToReal()</td>
<td></td>
</tr>
<tr>
<td>OCIRawToRealArray()</td>
<td></td>
</tr>
<tr>
<td>OCIRawToText()</td>
<td></td>
</tr>
<tr>
<td>OCIRawToText()</td>
<td></td>
</tr>
<tr>
<td>OCIRawToText()</td>
<td></td>
</tr>
<tr>
<td>OCIRawToText()</td>
<td></td>
</tr>
<tr>
<td>OCIRowid ROWID descriptor</td>
<td>2-11</td>
</tr>
<tr>
<td>OCIRowidToChar()</td>
<td>17-173</td>
</tr>
<tr>
<td>OCIsessionBegin()</td>
<td>2-11, 8-7, 16-30</td>
</tr>
<tr>
<td>OCIsessionEnd()</td>
<td>16-33</td>
</tr>
<tr>
<td>OCIsessionEnd()</td>
<td>16-33</td>
</tr>
<tr>
<td>OCIsessionGet()</td>
<td>16-34</td>
</tr>
<tr>
<td>OCIsessionGet()</td>
<td>16-34</td>
</tr>
<tr>
<td>OCIsessionPoolCreate()</td>
<td>16-40</td>
</tr>
<tr>
<td>OCIsessionPoolDestroy()</td>
<td>16-43</td>
</tr>
<tr>
<td>OCIsessionRelease()</td>
<td>16-44</td>
</tr>
<tr>
<td>OCIsessionRelease()</td>
<td>16-44</td>
</tr>
<tr>
<td>OCISessionBegin()</td>
<td>2-18, 2-31, 8-7, 16-30</td>
</tr>
<tr>
<td>OCISessionEnd()</td>
<td>16-33</td>
</tr>
<tr>
<td>OCISessionEnd()</td>
<td>16-33</td>
</tr>
<tr>
<td>OCISessionGet()</td>
<td>16-34</td>
</tr>
<tr>
<td>OCISessionPoolCreate()</td>
<td>16-40</td>
</tr>
<tr>
<td>OCISessionPoolDestroy()</td>
<td>16-43</td>
</tr>
<tr>
<td>OCISessionRelease()</td>
<td>16-44</td>
</tr>
<tr>
<td>OCISessionRelease()</td>
<td>16-44</td>
</tr>
<tr>
<td>OCISetExecute()</td>
<td>17-3</td>
</tr>
<tr>
<td>prefetch during</td>
<td>4-6</td>
</tr>
<tr>
<td>use of iters parameter</td>
<td>4-6</td>
</tr>
<tr>
<td>OCISetFetch()</td>
<td>E-8</td>
</tr>
<tr>
<td>OCISetFetch2()</td>
<td>4-15, 17-6</td>
</tr>
<tr>
<td>OCISetGetBindInfo()</td>
<td>16-89</td>
</tr>
<tr>
<td>OCISetGetPieceInfo()</td>
<td>17-8</td>
</tr>
<tr>
<td>OCISetPrepare()</td>
<td>17-10</td>
</tr>
</tbody>
</table>
OCIStmtPrepare2(), 17-12
preparing SQL statements, 4-3
OCIStmtRelease(), 17-14
OCIStmtSetPieceInfo(), 17-15
OCIString, 12-12
binding and defining, 12-12, 12-28
manipulation usage example, 12-12
OCIStringAllocSize(), 19-151
OCIStringAssign(), 19-152
OCIStringAssignText(), 19-153
OCIStringPtr(), 19-154
OCIStringResize(), 19-155
OCIStringSize(), 19-156
OCISubscriptionDisable(), 17-99
OCISubscriptionEnable(), 17-100
OCISubscriptionPost(), 17-101
OCISubscriptionRegister(), 17-103
OCISubscriptionUnRegister(), 17-105
OCITable, 12-14
binding and defining, 12-14, 12-28
OCITableDelete(), 19-158
OCITableExists(), 19-159
OCITableFirst(), 19-160
OCITableLast(), 19-161
OCITableNext(), 19-162
OCITablePrev(), 19-163
OCITableSize(), 19-164
OCITerminate(), 16-46
OCIThread package, 8-26
OCIThreadClose(), 17-122
OCIThreadCreate(), 17-123
OCIThreadDestroy(), 17-124
OCIThreadGet(), 17-128
OCIThreadInit(), 17-126
OCIThreadKeyDestroy(), 17-137
OCIThreadKeyGet(), 17-138
OCIThreadKeyInit(), 17-139
OCIThreadKeySet(), 17-140
OCIThreadMutexAcquire(), 17-141
OCIThreadMutexDestroy(), 17-142
OCIThreadMutexInit(), 17-143
OCIThreadMutexRelease(), 17-144
OCIThreadProcessInit(), 17-145
OCIThreadTerm(), 17-146
OCITransCommit(), 17-148
OCITransDetach(), 17-151
OCITransForget(), 17-152
OCITransMultiPrepare(), 8-6, 17-153
OCITransPrepare(), 17-154
OCITransRollback(), 17-155
OCITransStart(), 17-156
OCIType
description, 12-20
OCITypeAddAttr(), 21-4
OCITypeArrayByName(), 18-58
OCITypeArrayByRef(), 18-60
OCITypeBeginCreate(), 21-5
OCITypeByName(), 18-62
OCITypeByRef(), 18-64
OCITypeCode, 3-26
OCITypeElem
description, 12-20
OCITypeEndCreate(), 21-6
OCITypeMethod
description, 12-20
OCITypeSetBuiltin(), 21-7
OCITypeSetCollection(), 21-8
OCIUnicodeToCharSet(), 22-28
OCIUnicodeToCharSet(), 22-29
OCIUserCallbackGet(), 17-177
OCIUserCallbackRegister(), 17-179
OCIWideChar data type, 2-33
OCIWideCharInSizeToMultiByte(), 22-28
OCIWideCharIsAlnum(), 22-45
OCIWideCharIsAlpha(), 22-46
OCIWideCharIsChar(), 22-47
OCIWideCharIsDigit(), 22-48
OCIWideCharIsGraph(), 22-49
OCIWideCharIsLower(), 22-50
OCIWideCharIsPrint(), 22-51
OCIWideCharIsPunct(), 22-52
OCIWideCharIsSingleByte(), 22-53
OCIWideCharIsSpace(), 22-54
OCIWideCharIsUpper(), 22-55
OCIWideCharIsXdigit(), 22-56
OCIWideCharMultiByteLength(), 22-29
OCIWideCharStrCaseConversion(), 22-30
OCIWideCharStrcat(), 22-31
OCIWideCharStrchr(), 22-32
OCIWideCharStrncmp(), 22-33
OCIWideCharStrncpy(), 22-34
OCIWideCharStrlcn(), 22-35
OCIWideCharStrocat(), 22-36
OCIWideCharStrncpy(), 22-37
OCIWideCharStrncpy(), 22-39
OCIWideCharStrrchr(), 22-40
OCIWideCharToLower(), 22-41
OCIWideCharToUpper(), 22-42
OCIWideCharToUpper(), 22-43
OCIXmlDbFreeXmlCtx(), 14-18, 23-7
ocixmlb.h header file, 14-18
OCIXmlDbInitXmlCtx(), 14-18, 23-8
OID
See object identifiers
opaque, definition of, 1-2
optimistic locking
implementing, 14-11
ORA_EDITION environment variable, 8-23
ORA_NCHAR_LITERAL_REPLACE, 16-15
ORA-25219 error during enqueue, 9-52
Index-21
Oracle Call Interface
See OCI
Oracle data types, 3-1
  mapping to C, 12-2
Oracle Database
  transaction processing monitor, D-3
Oracle RAC, 9-12
Oracle Real Application Clusters, 9-12
Oracle Streams Advanced Queuing
  dequeue function, 17-89
  description, 9-47
  descriptor attributes, A-41
  enqueue function, 17-93
  functions, 17-88, E-26
OCI and, 9-47
OCI descriptors for, 9-48
OCI functions for, 9-48
OCI versus PL/SQL, 9-49
publish-subscribe notification in OCI, 9-54
Oracle XA Library
  additional documentation, D-5
  compiling and linking an OCI program, D-4
  dynamic registration, D-4
Oracle XML DB OCI functions, 14-17, 23-2
orab8 data type, 7-5
oratypes.h
  as parameter source in OCI, 3-28
  contents, 3-28
  definitions in, 3-28
oraub8 data type, 7-5
ORE
See object runtime environment
OTT
See OTT utility
OTT parameters, 15-20, 15-21
  CASE, 15-24
  CODE, 15-22
  CONFIG, 15-23
  ERRTYPE, 15-23
  FILE, 15-23
  INITFILE, 15-22
  INITFUNC, 15-23
  INTYPE, 15-22
  OUTTYPE, 15-22
  SCHEMA_NAMES, 15-24
  TRANSITIVE, 15-24
  URL, 15-24
  USERID, 15-21
  where they appear, 15-25
OTT utility
  command line, 15-4, D-5
  command-line syntax, 15-20
  creating types in the database, 15-3
  data type mappings, 15-8
  initialization function
    calling, 15-18
    tasks of, 15-19
    intype file, 15-25
    outtype file, 15-15
    overview, 15-1
  parameters, 15-21
  See OTT parameters
  providing an intype file, 15-6
  reference, 15-19
  restriction, 15-31
  sample output, 11-6
  use with OCI, 11-5
  using, 15-1, 15-2
  ottcfg.cfg, D-2
  outtype file, 15-25
    when running OTT utility, 15-15
  OUTTYPE OTT parameter, 15-22
  parameters
    attributes, 6-4
    object, 12-20
  parameters
    attributes, 6-4
    buffer lengths, 16-2
    modes, 16-1
    passing strings, 2-23
    string length, 16-2
  password management, 8-7, 8-9
  persistent objects, 11-3
  meta-attributes, 11-12
  piecewise
    binds and defines for LOBs, 5-39
    fetch, 5-38
  piecewise operations, 5-35
    fetch, 5-33, 5-38
    in PL/SQL, 5-37
    insert, 5-33
    update, 5-33
    valid data types, 5-34
  pin count, 11-21
  pin duration
    example, 14-12
    of objects, 14-11
  pinning objects, 14-5
  placeholders
    rules, 4-5
  PL/SQL, 1-6
    binding and defining nested tables, 5-32
    binding and defining REF CURSORS, 5-32
    binding placeholders, 2-29
    defining output variables, 5-17
    piecewise operations, 5-37
    uses in OCI applications, 2-29
    using in OCI applications, 2-29
    using in OCI programs, 5-5
  polling mode, 2-27
  positioned deletes, 2-25
  positioned updates, 2-25
  prefetching
    during OCIStmtExecute(), 4-6
  P
  packages
    attributes, 6-6
    describing, 6-1
  parameter descriptor, 2-11
  attributes, 6-4, A-39
  object, 12-20
  parameters
    attributes, 6-4
    buffer lengths, 16-2
    modes, 16-1
    passing strings, 2-23
    string length, 16-2
  password management, 8-7, 8-9
  persistent objects, 11-3
  meta-attributes, 11-12
  piecewise
    binds and defines for LOBs, 5-39
    fetch, 5-38
  piecewise operations, 5-35
    fetch, 5-33, 5-38
    in PL/SQL, 5-37
    insert, 5-33
    update, 5-33
    valid data types, 5-34
  pin count, 11-21
  pin duration
    example, 14-12
    of objects, 14-11
  pinning objects, 14-5
  placeholders
    rules, 4-5
  PL/SQL, 1-6
    binding and defining nested tables, 5-32
    binding and defining REF CURSORS, 5-32
    binding placeholders, 2-29
    defining output variables, 5-17
    piecewise operations, 5-37
    uses in OCI applications, 2-29
    using in OCI applications, 2-29
    using in OCI programs, 5-5
  polling mode, 2-27
  positioned deletes, 2-25
  positioned updates, 2-25
  prefetching
    during OCIStmtExecute(), 4-6
LOBs, 7-19
setting prefetch memory size, 4-13
setting row count, 4-13
preparing multiple branches in a single message, 8-6
procedures
attributes, 6-6
process
handle attributes, A-75
proxy access, 2-15
proxy authentication, 2-15, 8-13
publish-subscribe
_SYSTEM_TRIG_ENABLED parameter, 9-64
COMPATIBLE parameter, 9-55
example, 9-64
functions, 9-55, 17-88, E-26
handle attributes, 9-56, A-51
LDAP registration, 9-59
notification callback, 9-61
notification in OCI, 9-54
subscription handle, 9-56
query
explicit describe, 4-11
See SQL query
RAW
external data type, 3-13
read-only parameter descriptor, 2-9
REF
external data type, 3-16
REF columns
direct path loading of, 13-21
REF CURSOR variables
binding and defining, 5-32
references to objects
See REFS
refreshing, 14-9
objects, 14-9
RSFs, D-1
rule sets
attributes, 6-16
type OCI_PTYPE_RULE_SET, 6-16
rules
attributes, 6-15
type OCI_PTYPE_RULE, 6-15
running OCI application, D-3
sample programs, B-1, D-2
samples directory, D-2
sb1
definition, 3-28
sb2
definition, 3-28
sb4
definition, 3-28
scatter/gather for binds/defines, 5-20
schema type attributes
type OCI_PTYPE_SCHEMA, 6-14
SCHEMA_NAMES OTT parameter, 15-24
usage, 15-29
SCHEMA.QUEUE, 17-104
SCHEMA.QUEUE:CONSUMER_NAME, 17-104
schemas
attributes, 6-14, 6-15, 6-16, 6-17
describing, 6-1
scripts
authenticating users in, 8-9
scrollable cursor
  example, 4-15
  in OCI, 4-14
  increasing performance, 4-15
SDK for Instant Client Light, 1-27
secondary memory
  of object, 14-13
SECUREFILE parameter, 7-22
SecureFiles, 7-22
  OCILOBGetContentType(), 17-56
  OCILOBSetContentType(), 17-77
SecureFiles LOBs, 7-22
security
  init.ora parameters, 8-23
  OCI enhancements, 8-23
select list
  describing, 4-9
sequences
  attributes, 6-11
  describing, 6-1
server handle
  attributes, A-11
  description, 2-5
  setting in service context, 2-6
server round-trips
  cache functions, C-4
  data type mapping and manipulation functions, C-6
  definition of, C-1
  describe operation, C-5
  LOB functions, C-3
  object functions, C-4
  relational functions, C-6
service context handle
  attributes, A-8
  description, 2-5
  elements of, 2-5
session creation, 2-14
session management, 8-7, 8-9
session migration, 8-8, 16-31
session pool handle
  attributes, A-24
  session pooling, 9-7, 9-23
  example, 9-12
  functionality, 9-8
  runtime connection load balancing, 9-12
  tagging, 9-8
shutting down databases, 10-17
signal handler, 8-24
skip parameters
  for arrays of structures, 5-18
  for standard arrays, 5-19
snapshot descriptor, 2-9, 2-10
snapshots
  executing against, 4-6
SQL query
  binding placeholders
    See bind operations
  defining output variables, 4-12, 5-13, 12-26
    See define operations
  fetching results, 4-13
  statement type, 1-6
SQL statements, 1-4
  binding placeholders in, 4-4, 5-1, 12-25
  determining type prepared, 4-3
  executing, 4-5
  preparing for execution, 4-3
  processing, 4-1
  types
    control statements, 1-5
    data definition language, 1-5
    data manipulation language, 1-5
    embedded SQL, 1-7
    PL/SQL, 1-6
    queries, 1-6
sqlnet.ora, 8-23
sqlnet.ora, controlling ADR, 10-35
SQLT typecodes, 3-27
SQLT_BDOUBLE, 3-20
SQLT_BFLOAT, 3-20
SQLT_CHR, A-63
SQLT_REF
  definition, 3-16
  description, 3-16
starting up databases, 10-17
STATEFUL, 9-22
stateful sessions, 9-2
STATELESS, 9-22
stateless sessions, 9-2
statement caching, 9-26
  code example, 9-29
statement handle
  attributes, A-27
  description, 2-6
statically linked applications, 1-12
stored functions
  describing, 6-1
stored procedures
  describing, 6-1
STRING
  external data type, 3-11
strings
  passing as parameters, 2-23
structures
  arrays of, 5-18
  subprogram attributes, 6-6
subscription handle, 2-7
  attributes, A-51
supporting UTF-16 Unicode in OCI, 2-30, 2-32, 2-33,
sword
definition in oratypes.h file, 3-28
synonyms
attributes, 6-10
describing, 6-1

T

table alias
attributes
type OCI_PTYPE_TABLE_ALIAS, 6-16
table change descriptor, 2-9
tables
attributes, 6-5
describing, 6-1
limitations on OCIDescribeAny() and
OCISqlExecCt(), 6-2
TAF (transparent application failover)
callback registration, 9-40
callbacks, 9-37
connections enabled by, 9-47
in connection pools, 9-3
OCI callbacks, 9-37
tagging
custom pooling, 9-46
session pooling, 9-8, 16-37, 16-44
TDO
definition, 12-25
description, 12-20
in the object cache, 14-17
obtaining, 12-20
terminology
navigational functions, 18-3
used in this manual, 1-7
thread handle description, 2-7, 8-30
thread management functions, 17-121
thread safety, 8-24
advantages of, 8-25
basic concepts, 8-24
implementing with OCI, 8-25
mixing 7.x and 8.0 calls, 8-26
required OCI calls, 8-25
three-tier architectures, 8-25
threads package, 8-26
three-tier architectures
thread safety, 8-25
time zone files differ on client and server, 10-36
TIMESTAMP data type, 3-19
TIMESTAMP descriptor, 2-9
TIMESTAMP WITH LOCAL TIME ZONE data
type, 3-19
TIMESTAMP WITH LOCAL TIME ZONE
descriptor, 2-9
TIMESTAMP WITH TIME ZONE data type, 3-19
TIMESTAMP WITH TIME ZONE descriptor, 2-9
TimesTen In-Memory Database access from
OCI, 3-1i, 1-1, 10-1
TNS_ADMIN, 1-22
tnsnames.ora file, 1-17
top-level memory
of object, 14-13
transaction handle
attributes, A-26
description, 2-6
transaction identifier, 8-2
transaction processing monitor
additional documentation, D-5
interacting with Oracle Database, D-3
types, D-3
transactional complexity
levels in OCI, 8-2
transactions
committing, 2-19
functions, 17-147
global, 8-2
branch states, 8-4
branches, 8-3
one-phase commit, 8-5
transactions identifier, 8-2
two-phase commit, 8-5
global examples, 8-6
initialization parameters, 8-6
local, 8-2
OCI functions for
transactions, 8-1
read-only, 8-2
rolling back, 2-19
serializable, 8-2
transient objects, 11-4
LOBs
attributes, 7-3
meta-attributes, 11-14
TRANSITIVE OTT parameter, 15-7, 15-11, 15-24
transparent application failover
See TAF
type attributes
attributes, 6-8
type descriptor object, 11-5, 12-20
type evolution, 11-30
object cache, 14-17
type functions
attributes, 6-9
type inheritance
OTT utility support, 15-13
type method attributes, 6-9
type procedures
attributes, 6-9
type reference, 11-25
typecodes, 3-25
types
attributes, 6-7
describing, 6-1

U

ub1
definition, 3-28
ub2
definition, 3-28
ub4
  definition, 3-28
Unicode
data buffer alignment, 2-35
  OCILobRead(), E-17
  OCILobWrite(), 17-31, 17-84, E-22
Universal ROWID, 3-5
unmarking objects, 14-8
unpinning objects, 11-21, 14-7
UNSIGNED
  external data type, 3-14
updates
  piecewise, 5-33, 5-35
  positioned, 2-25
upgrading
  7.x to 8.0, 1-15
  7.x to 8.0 OCI, 1-15
upgrading OCI, 1-12
URL OTT parameter, 15-24
UROWID
  Universal ROWID, 3-5
user callback descriptor, 2-9
user memory
  allocating, 2-12
user session handle
  attributes, A-14
  description, 2-5
  setting in service context, 2-6
user-defined callback functions, 9-29
  registering, 9-30
USERID OTT parameter, 15-21
using an explicit describe on a named collection type
  example, 6-23
using an explicit describe on a named object type
  example, 6-22
using an explicit describe to retrieve column data
  types for a table
  example, 6-19
utext
  Unicode data type, 5-27, 5-32
UTF-16 data, sample code, 5-31

V
  values, 11-3
    in object applications, 11-4
VARCHAR
  external data type, 3-12
VARCHAR2
  external data type, 3-7
variable type attributes
  type OCI_PTYPE_VARIABLE_TYPE, 6-17
VARNUM
  external data type, 3-12
VARRAW
  external data type, 3-14
varrays, 3-16, 12-14
  as objects, 11-1
  C mapping, 15-9
define calls, 12-26
direct path loading not supported, 13-7
iterator, 19-17
NULLs, 11-23
or collection Iterator example, 12-15
scan, 19-20
version compatibility, 1-12
views
  attributes, 6-5
  describing, 6-1

W
  wchar_t data type, 2-33, 5-32, 22-14
with_context
  argument to external procedure functions, 20-2

X
  XA Library
    compiling and linking an OCI program, D-4
    functions, D-3
    OCI support, 1-11
    overview, D-3
    XA specification, 8-3
    xa.h header, 1-11
    XID
      See transaction identifier
    XML DB functions, 14-18, 23-2
    XML support in OCI, 14-17, 23-1
    xtramem_sz parameter
      using, 2-12