Oracle Forms Developer and Oracle Reports Developer: Common Built-in Packages, Release 6

The part number for this volume is A73152-01

Copyright © 1999, 2000, Oracle Corporation. All rights reserved.

Portions copyright © Blue Sky Software Corporation. All rights reserved.

Contributors: Fred Bethke, Joan Carter, Kenneth Chu, Kate Dumont, Tom Haunert, Colleen McCann, Leanne Soylemez, Poh Lee Tan, Tony Wolfram

The programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be licensee's responsibility to take all appropriate fail-safe, back up, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle disclaims liability for any damages caused by such use of the Programs.

This Program contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright, patent and other intellectual property law. Reverse engineering of the software is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If this Program is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication and disclosure of the Programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle is a registered trademark, and Express, Oracle Browser, Oracle Forms, Oracle Graphics, Oracle Installer, Oracle Reports, Oracle7, Oracle8, Oracle Web Application Server, Personal Oracle, Personal Oracle Lite, and PL/SQL are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.
# Table of Contents

**BUILT-IN PACKAGES** ................................................................. 1  
  ABOUT BUILT-IN PACKAGES ....................................................... 1  
  ABOUT THE DDE PACKAGE ...................................................... 3  
  MICROSOFT WINDOWS PREDEFINED DATA FORMATS .................. 4  
  DDE PREDEFINED EXCEPTIONS ................................................... 6  
  ABOUT THE DEBUG PACKAGE ................................................... 8  
  ABOUT THE LIST PACKAGE ..................................................... 8  
  ABOUT THE OLE2 PACKAGE ...................................................... 8  
  ABOUT THE ORA_FFI PACKAGE ............................................... 8  
  ABOUT THE ORA_NLS PACKAGE .............................................. 9  
  ORA_NLS CHARACTER CONSTANTS ......................................... 9  
  ORA_NLS NUMERIC CONSTANTS .............................................. 11  
  ABOUT THE ORA_PROF PACKAGE ........................................... 12  
  ABOUT THE TEXT_IO PACKAGE ............................................. 12  
  ABOUT THE TOOL_ENV PACKAGE .......................................... 13  
  ABOUT THE TOOL_ERR PACKAGE ........................................... 13  
  ABOUT THE TOOL_RES PACKAGE .......................................... 14  
  BUILDING RESOURCE FILES .................................................. 14  
  ABOUT THE EXEC_SQL PACKAGE .......................................... 16  
  CONNECTION AND CURSOR HANDLES ...................................... 17  
  RETRIEVING RESULT SETS FROM QUERIES OR NON-ORACLE STORED PROCEDURES .......................................................... 17  
  EXEC_SQL PREDEFINED EXCEPTIONS ...................................... 18  
  USING THE EXEC_SQL PACKAGE ........................................... 19  
  EXECUTING ARBITRARY SQL AGAINST ANY CONNECTION ............... 19  
  COPYING DATA BETWEEN TWO DATABASES ................................. 21  
  EXECUTING A NON-ORACLE DATABASE STORED PROCEDURE AND FETCHING ITS RESULT SET ...................................................... 24  
  ALPHABETIC LIST OF PACKAGED SUBPROGRAMS ......................... 25

**DDE PACKAGE** ................................................................. 29  
  DDE PACKAGE ..................................................................... 29  
  DDE.APP_BEGIN ................................................................... 29  
  DDE.APP_END ..................................................................... 30  
  DDE.APP_FOCUS .................................................................. 31  
  DDE.EXECUTE ..................................................................... 32  
  DDE.GETFORMATNUM .......................................................... 32
We Appreciate Your Comments

Reader’s Comment Form - A73152-01

Oracle Corporation welcomes your comments about this manual’s quality and usefulness. Your feedback is an important part of our revision process.

• Did you find any errors?
• Is the information presented clearly?
• Are the examples correct? Do you need more examples?
• What features did you like?

If you found any errors or have any other suggestions for improvement, please send your comments by e-mail to oddoc@us.oracle.com.

Thank you for your help.
Preface

Welcome to Release 6i of the Oracle Forms Developer and Oracle Reports Developer: Common Built-in Packages.

This reference guide includes information to help you effectively work with Forms Developer and contains detailed information about its built-in packages.

This preface explains how this user’s guide is organized and introduces other sources of information that can help you use Forms Developer.
Prerequisites

You should be familiar with your computer and its operating system. For example, you should know the commands for deleting and copying files and understand the concepts of search paths, subdirectories, and path names. Refer to your Microsoft Windows 95 or NT and DOS product documentation for more information.

You should also understand the fundamentals of Microsoft Windows, such as the elements of an application window. You should also be familiar with such programs as the Explorer, Taskbar or Task Manager, and Registry.

Notational Conventions

The following typographical conventions are used in this guide:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-width font</td>
<td>Text in a fixed-width font indicates commands that you enter exactly as shown. Text typed on a PC is not case-sensitive unless otherwise noted. In commands, punctuation other than brackets and vertical bars must be entered exactly as shown.</td>
</tr>
<tr>
<td>lowercase</td>
<td>Lowercase characters in a command statement represent a variable. Substitute an appropriate value.</td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>Uppercase characters within the text represent command names, SQL reserved words, and keywords.</td>
</tr>
<tr>
<td>boldface</td>
<td>Boldface is used to indicate user interface items such as menu choices.</td>
</tr>
</tbody>
</table>
and buttons.

C> C> represents the DOS prompt. Your prompt may differ.

Related Publications

You may also wish to consult the following Oracle documentation:

<table>
<thead>
<tr>
<th>Title</th>
<th>Part Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle Forms Developer and Oracle Reports Developer: Guidelines for Building Applications</td>
<td>A73073</td>
</tr>
<tr>
<td>SQL*Plus User’s Guide and Reference Version 3.1</td>
<td>A24801</td>
</tr>
</tbody>
</table>
Built-in Packages

About built-in packages

Both Forms Developer and Reports Developer provide several built-in packages that contain many PL/SQL constructs you can reference while building applications or debugging your application code. These built-in packages are not installed as extensions to package STANDARD. As a result, any time you reference a construct in one of the packages, you must prefix it with the package name (for example, Text_IO.Put_Line). The built-in packages are:

- **DDE** provides Dynamic Data Exchange support within Forms Developer components.
- **Debug** provides procedures, functions, and exceptions for debugging PL/SQL program units.
- **EXEC_SQL** provides procedures and functions for executing dynamic SQL within PL/SQL code written for Forms Developer applications.
- **List** provides procedures, functions, and exceptions you can use to create and maintain lists of character strings (VARCHAR2). This provides a means of creating arrays in PL/SQL Version 1.
- **OLE2** provides a PL/SQL API for creating, manipulating, and accessing attributes of OLE2 automation objects.
- **Ora_Ffi** provides a public interface for calling out to foreign(C) functions from PL/SQL.
- **Ora_Nls** enables you to extract high-level information about your current language environment.
- **Ora_Prof** provides procedures, functions, and exceptions you can use for tuning your PL/SQL program units (e.g. examining how much time a specific piece of code takes to run).
- **Text_IO** provides constructs that allow you to read and write information from and to files.
- **Tool_Env** allows you to interact with Oracle environment variables.
- **Tool_Err** allows you to access and manipulate the error stack created by other built-in packages such as DEBUG.
- **Tool_Res** provides a means of extracting string resources from a resource file with the
goal of making PL/SQL code more portable by isolating all textual data in the resource file.

The following packages are used only internally by Forms Developer. There are no subprograms available for external use with these packages.

- **Ora_De**: Contains constructs used by Forms Developer for private PL/SQL services.
- **STPRO**: Used internally by Forms Developer to call subprograms stored in the database.
- **C**: Calls to this package are automatically generated.
About the DDE package

The DDE Package provides Dynamic Data Exchange (DDE) support within Forms Developer components.

Dynamic Data Exchange (DDE) is a mechanism by which applications can communicate and exchange data in Windows. DDE client support is added as a procedural extension to Forms Developer. The PL/SQL package for DDE support provides application developers with an Application Programming Interface (API) for accessing DDE functionality from within PL/SQL procedures and triggers.

The DDE functions enable Oracle applications to communicate with other DDE-compliant Windows applications (servers) in three ways:

- importing data
- exporting data
- executing commands against the DDE Server

In this release, DDE does not include the following:

- data linking (advise transaction)

Oracle applications cannot automatically receive an update notice when a data item has changed.

- Server support

Oracle applications cannot respond to commands or requests for data from a DDE client. Oracle Applications must initiate the DDE conversation (although data may still be transferred in either direction).

Support Functions

These functions are used to start and stop other DDE server applications.

Connect/Disconnect Functions

These functions are used to connect to and disconnect from DDE server applications.

Transaction Functions

These functions are used to exchange data with DDE server applications.

Datatype Translation Functions

These functions are used to translate DDE datatype constants to strings and back; in addition, DDE.Getformatnum allows users to register a new data format that is not predefined by Windows. Note that these functions do not translate the data itself (all DDE data is represented with the CHAR datatype in PL/SQL), just datatype constants.

Note: In previous releases of Forms Developer, it was necessary to attach a stub library so that calls to the Windows-specific DDE functions would compile and run correctly on
non-Windows platforms. This is no longer necessary. However, when you attempt to execute a Windows-specific built-in function on a non-Windows platform, the following messages are generated:
FRM-40735: Trigger <name> raised unhandled exception.
ORA-06509, 00000 PL/SQL: ICD vector missing for this package.

## Microsoft Windows predefined data formats

See the Exceptions section for predefined data format exceptions.

<table>
<thead>
<tr>
<th>Data Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDE.Cf_Bitmap</td>
<td>The data is a bitmap.</td>
</tr>
<tr>
<td>DDE.Cf_Dib</td>
<td>The data is a memory object containing a BITMAPINFO structure followed by the bitmap data.</td>
</tr>
<tr>
<td>DDE.Cf_Dif</td>
<td>The data is in Data Interchange Format (DIF).</td>
</tr>
<tr>
<td>DDE.Cf_Dspbitmap</td>
<td>The data is a bitmap representation of a private format. This data is displayed in bitmap format in lieu of the privately formatted data.</td>
</tr>
<tr>
<td>DDE.Cf_Dspmetafile-Pict</td>
<td>The data is a metafile representation of a private data format. This data is displayed in metafile-picture format in lieu of the privately formatted data.</td>
</tr>
<tr>
<td>DDE.CF_Dsptext</td>
<td>The data is a Textual Representation of a private data format. This data is displayed in Text Format in lieu of the privately formatted data.</td>
</tr>
<tr>
<td>DDE.Cf_Metafilepict</td>
<td>The data is a metafile.</td>
</tr>
<tr>
<td>DDE.Cf_Oemtext</td>
<td>The data is an array of Text Characters in the OEM character set. Each line ends with a carriage return-linefeed (CR-LF) combination. A null character signals the end of the data.</td>
</tr>
<tr>
<td>DDE.Cf_Owner-Display</td>
<td>The data is in a private format that the clipboard owner must display.</td>
</tr>
<tr>
<td>DDE.Cf_Palette</td>
<td>The data is a color palette.</td>
</tr>
<tr>
<td>DDE.Cf_Pendata</td>
<td>The data is for the pen extensions to the Windows operating system.</td>
</tr>
<tr>
<td>DDE.Cf_Riff</td>
<td>The data is in Resource Interchange File Format (RIFF).</td>
</tr>
<tr>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DDE.Cf_Sylk</td>
<td>The data is in Microsoft Symbolic Link (SYLK) format.</td>
</tr>
<tr>
<td>DDE.Cf_Text</td>
<td>The data is an array of Text Characters. Each line ends with a carriage return-linefeed (CR-LF) combination. A null character signals the end of the data.</td>
</tr>
<tr>
<td>DDE.Cf_Tiff</td>
<td>The data is in Tagged Image File Format (TIFF).</td>
</tr>
<tr>
<td>DDE.Cf_Wave</td>
<td>The data describes a sound wave. This is a subset of the CF_RIFF data format; it can be used only for RIFF WAVE files.</td>
</tr>
</tbody>
</table>
DDE predefined exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDE.DDE_App_Failure</td>
<td>An application program specified in a DDE.App_Begin call could not be started.</td>
</tr>
<tr>
<td>DDE.DDE_App_Not_Found</td>
<td>An application ID specified in a DDE.App_End or DDE.App_Focus call does not correspond to an application that is currently running.</td>
</tr>
<tr>
<td>DDE.DDE_FMT_Not_Found</td>
<td>A format number specified in a DDE.GetFormatstr call is not known.</td>
</tr>
<tr>
<td>DDE.DDE_FMT_Not_Reg</td>
<td>A format string specified in a DDE.GetFormatnum call does not correspond to a predefined format and could not be registered as a user-defined format.</td>
</tr>
<tr>
<td>DDE.DDE_Init_Failed</td>
<td>The application was unable to initialize DDE communications, which caused a call to the DDE Layer to fail.</td>
</tr>
<tr>
<td>DDE.DDE_Param_Err</td>
<td>An invalid parameter, such as a NULL value, was passed to a DDE Package routine.</td>
</tr>
<tr>
<td>DDE.Dmlerr_Busy</td>
<td>A transaction failed because the server application was busy.</td>
</tr>
<tr>
<td>DDE.Dmlerr_Dataacktimeout</td>
<td>A request for a synchronous data transaction has timed out.</td>
</tr>
<tr>
<td>DDE.Dmlerr_Execacktimeout</td>
<td>A request for a synchronous execute transaction has timed out.</td>
</tr>
<tr>
<td>DDE.Dmlerr_Invalidparameter</td>
<td>A parameter failed to be validated. Some of the possible causes are as follows: The application used a data handle initialized with a different item-name handle or clipboard data format than that required by the transaction. The application used an invalid data handle.</td>
</tr>
</tbody>
</table>
conversation identifier.

More than one instance of the application used the same object.

DDE.Dmlerr_Memory_Error A memory allocation failed.

DDE.Dmlerr_No_Conv_Est A client’s attempt to establish a conversation has failed. The service or topic name in a DDE.Initiate call may be in error.

DDE.Dmlerr_NotProcessed A transaction failed. The item name in a DDE.Poke or DDE.Request transaction may be in error.

DDE.Dmlerr_NotSupported A call is made to the DDE package but DDE is not supported on the current software platform.

DDE.Dmlerr_Pokeacktimeout A request for a synchronous DDE.Poke transaction has timed out.

DDE.Dmlerr_Postmsg_Failed An internal call to the PostMessage function has failed.

DDE.Dmlerr_Server_Died The server terminated before completing a transaction.

DDE.Dmlerr_Sys_Error An internal error has occurred in the DDE Layer.
About the Debug package

The Debug package contains procedures, functions, and exceptions for use when debugging your PL/SQL program units. Use these built-in subprograms to create debug triggers and set breakpoints with triggers.

About the List package

The List package contains procedures, functions, and exceptions you can use to create and maintain lists of character strings (VARCHAR2). These services provide a means of creating arrays in PL/SQL Version 1.

About the OLE2 package

The OLE2 package provides a PL/SQL API for creating, manipulating, and accessing attributes of OLE2 automation objects. OLE2 automation objects encapsulate a set of attributes and methods that can be manipulated or invoked from an OLE2 automation client. The OLE2 package allows users to access OLE2 automation servers directly from PL/SQL. Refer to the OLE2 programmers documentation for each OLE2 automation server for the object types, methods, and syntax specification.

About the Ora_Ffi package

The Ora_Ffi package provides a foreign function interface for invoking C functions in a dynamic library. Note that float arguments must be converted to doubles. If you must use ANSI declarations, use only doubles within your code.
About the Ora_NLS package

The Ora_Nls package enables you to extract high-level information about your current language environment. This information can be used to inspect attributes of the language, enabling you to customize your applications to use local date and number format. Information about character set collation and the character set in general can also be obtained.

Facilities are also provided for retrieving the name of the current language and character set, allowing you to create applications that test for and take advantage of special cases.

Ora_NLS character constants

Use the following constants to retrieve character information about the current language. All of the constants are of type PLS_INTEGER, with each assigned an integer value.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>day1</td>
<td>full name of day 1</td>
<td>1  sunday</td>
</tr>
<tr>
<td>day2</td>
<td>full name of day 2</td>
<td>2  monday</td>
</tr>
<tr>
<td>day3</td>
<td>full name of day 3</td>
<td>3  tuesday</td>
</tr>
<tr>
<td>day4</td>
<td>full name of day 4</td>
<td>4  wednesday</td>
</tr>
<tr>
<td>day5</td>
<td>full name of day 5</td>
<td>5  thursday</td>
</tr>
<tr>
<td>day6</td>
<td>full name of day 6</td>
<td>6  friday</td>
</tr>
<tr>
<td>day7</td>
<td>full name of day 7</td>
<td>7  saturday</td>
</tr>
<tr>
<td>day1_abbr</td>
<td>abbr. name of day 1</td>
<td>8  sun</td>
</tr>
<tr>
<td>day2_abbr</td>
<td>abbr. name of day 2</td>
<td>9  mon</td>
</tr>
<tr>
<td>day3_abbr</td>
<td>abbr. name of day 3</td>
<td>10 tue</td>
</tr>
<tr>
<td>day4_abbr</td>
<td>abbr. name of day 4</td>
<td>11 wed</td>
</tr>
<tr>
<td>day5_abbr</td>
<td>abbr. name of day 5</td>
<td>12 thu</td>
</tr>
<tr>
<td>day6_abbr</td>
<td>abbr. name of day 6</td>
<td>13 fri</td>
</tr>
<tr>
<td>day7_abbr</td>
<td>abbr. name of day 7</td>
<td>14 sat</td>
</tr>
<tr>
<td>mon1</td>
<td>full name of month 1</td>
<td>15 january</td>
</tr>
<tr>
<td>mon2</td>
<td>full name of month 2</td>
<td>16 february</td>
</tr>
<tr>
<td>mon3</td>
<td>full name of month 3</td>
<td>17 march</td>
</tr>
<tr>
<td>mon4</td>
<td>full name of month 4</td>
<td>18 april</td>
</tr>
<tr>
<td>mon5</td>
<td>full name of month 5</td>
<td>19 may</td>
</tr>
<tr>
<td>mon6</td>
<td>full name of month 6</td>
<td>20 june</td>
</tr>
<tr>
<td>mon7</td>
<td>full name of month 7</td>
<td>21 july</td>
</tr>
<tr>
<td>mon8</td>
<td>full name of month 8</td>
<td>22 aug</td>
</tr>
<tr>
<td>mon9</td>
<td>full name of month 9</td>
<td>23 sept</td>
</tr>
<tr>
<td>mon10</td>
<td>full name of month 10</td>
<td>24 oct</td>
</tr>
<tr>
<td>mon11</td>
<td>full name of month 11</td>
<td>25 nov</td>
</tr>
<tr>
<td>variable</td>
<td>description</td>
<td>value</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>mon12</td>
<td>full name of month 12</td>
<td></td>
</tr>
<tr>
<td>mon1_abbr</td>
<td>abbr. name of month 1</td>
<td>27</td>
</tr>
<tr>
<td>mon2_abbr</td>
<td>abbr. name of month 2</td>
<td>28</td>
</tr>
<tr>
<td>mon3_abbr</td>
<td>abbr. name of month 3</td>
<td>29</td>
</tr>
<tr>
<td>mon4_abbr</td>
<td>abbr. name of month 4</td>
<td>30</td>
</tr>
<tr>
<td>mon5_abbr</td>
<td>abbr. name of month 5</td>
<td>31</td>
</tr>
<tr>
<td>mon6_abbr</td>
<td>abbr. name of month 6</td>
<td>32</td>
</tr>
<tr>
<td>mon7_abbr</td>
<td>abbr. name of month 7</td>
<td>33</td>
</tr>
<tr>
<td>mon8_abbr</td>
<td>abbr. name of month 8</td>
<td>34</td>
</tr>
<tr>
<td>mon9_abbr</td>
<td>abbr. name of month 9</td>
<td>35</td>
</tr>
<tr>
<td>mon10_abbr</td>
<td>abbr. name of month 10</td>
<td>36</td>
</tr>
<tr>
<td>mon11_abbr</td>
<td>abbr. name of month 11</td>
<td>37</td>
</tr>
<tr>
<td>mon12_abbr</td>
<td>abbr. name of month 12</td>
<td>38</td>
</tr>
<tr>
<td>yes_str</td>
<td>Affirmative response for queries</td>
<td>39</td>
</tr>
<tr>
<td>no_str</td>
<td>Negative response for queries</td>
<td>40</td>
</tr>
<tr>
<td>am_str</td>
<td>Local equivalent of AM</td>
<td>41</td>
</tr>
<tr>
<td>pm_str</td>
<td>Local equivalent of PM</td>
<td>42</td>
</tr>
<tr>
<td>ad_str</td>
<td>Local equivalent of AD</td>
<td>43</td>
</tr>
<tr>
<td>bc_str</td>
<td>Local equivalent of BC</td>
<td>44</td>
</tr>
<tr>
<td>decimal</td>
<td>Decimal character</td>
<td>45</td>
</tr>
<tr>
<td>groupsep</td>
<td>Group separator</td>
<td>46</td>
</tr>
<tr>
<td>int_currency</td>
<td>Int. currency symbol</td>
<td>47</td>
</tr>
<tr>
<td>local_currency</td>
<td>Local currency symbol</td>
<td>48</td>
</tr>
<tr>
<td>local_date_fmt</td>
<td>Local date format</td>
<td>49</td>
</tr>
<tr>
<td>local_timeFmt</td>
<td>Local time format</td>
<td>50</td>
</tr>
<tr>
<td>default_date_fmt</td>
<td>Oracle Default date format</td>
<td>51</td>
</tr>
<tr>
<td>default_time_fmt</td>
<td>Oracle Default time format</td>
<td>52</td>
</tr>
<tr>
<td>language</td>
<td>Language name</td>
<td>53</td>
</tr>
<tr>
<td>language_abbr</td>
<td>ISO abbreviation for language</td>
<td>54</td>
</tr>
<tr>
<td>character_set</td>
<td>Default character set name</td>
<td>55</td>
</tr>
<tr>
<td>territory</td>
<td>Default territory name</td>
<td>56</td>
</tr>
<tr>
<td>current_decimal</td>
<td>Current decimal character</td>
<td>57</td>
</tr>
<tr>
<td>current_groupsep</td>
<td>Current group separator</td>
<td>58</td>
</tr>
<tr>
<td>current_currency</td>
<td>Current local currency</td>
<td>59</td>
</tr>
<tr>
<td>current_date_fmt</td>
<td>Current Oracle Date format</td>
<td>60</td>
</tr>
<tr>
<td>current_language</td>
<td>Current language</td>
<td>70</td>
</tr>
<tr>
<td>current_territory</td>
<td>Current territory</td>
<td>61</td>
</tr>
<tr>
<td>current_character_set</td>
<td>Current character set</td>
<td>62</td>
</tr>
</tbody>
</table>
## Ora_NLS numeric constants

Use the following constants to retrieve numeric information about the current language. All of the constants are of type PLS_INTEGER, with each assigned an integer value.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal_places</td>
<td>Currency Decimal Places</td>
<td>63</td>
</tr>
<tr>
<td>sign_placement</td>
<td>Sign location: 0=before, 1=after</td>
<td>64</td>
</tr>
<tr>
<td>initcap_month</td>
<td>Initcap month names: 0=NO,1=YES</td>
<td>65</td>
</tr>
<tr>
<td>initcap_day</td>
<td>Initcap day names: 0=NO,1=YES</td>
<td>66</td>
</tr>
<tr>
<td>week_start</td>
<td>Week start day: 0=sunday</td>
<td>67</td>
</tr>
<tr>
<td>week_num_calc</td>
<td>Week num calc: 1=ISO, 0=non ISO</td>
<td>68</td>
</tr>
<tr>
<td>iso_alphabet</td>
<td>Current ISO alphabet number</td>
<td>69</td>
</tr>
</tbody>
</table>
About the Ora_Prof package

The Ora_Prof package contains procedures, functions, and exceptions you can use when tuning your PL/SQL program units. The services in this package allow you to track the amount of time pieces of your code take to run.

About the Text_IO package

The Text_IO Package contains constructs that provide ways to write and read information to and from files. There are several procedures and functions available in Text_IO, falling into the following categories:

file operations
- The FILE_TYPE record, the FOPEN and IS_OPEN functions, and the FCLOSE procedure enable you to define FILE_TYPE variables, open files, check for open files, and close open files, respectively.

output (write) operations
- The PUT, PUTF, PUT_LINE, and NEW_LINE procedures enable you to write information to an open file or output it to the Interpreter.

input (read) operations
- The GET_LINE procedure enables you to read a line from an open file.
Using Text_IO constructs example

Below is an example of a procedure that echoes the contents of a file. Notice that the procedure includes several calls to Text_IO constructs:

```
PROCEDURE echo_file_contents IS
  in_file   Text_IO.File_Type;
  linebuf   VARCHAR2(80);
BEGIN
  in_file := Text_IO.Fopen('echo.txt', 'r');
  LOOP
    Text_IO.Get_Line(in_file, linebuf);
    Text_IO.Put(linebuf);
    Text_IO.New_Line;
  END LOOP;
EXCEPTION
  WHEN no_data_found THEN
    Text_IO.Put_Line('Closing the file...');
    Text_IO.Fclose(in_file);
END;
```

About the Tool_Env package

The Tool_Env package allows you to interact with Oracle environment variables by retrieving their values for use in subprograms.

About the Tool_Err package

In addition to using exceptions to signal errors, some built-in packages (e.g., the Debug package) provide additional error information. This information is maintained in the form of an "error stack".

The error stack contains detailed error codes and associated error messages. Errors on the stack are indexed from zero (oldest) to n-1 (newest), where n is the number of errors currently on the stack. Using the services provided by the Tool_Err package, you can access and manipulate the error stack.

Using Tool_Err constructs example

The following procedure shows how you can use constructs within the Tool_Err package to handle errors generated by the Debug.Interpret built-in:

```
PROCEDURE error_handler IS
  /* Call a built-in that interprets a command */
```
BEGIN
  Debug.Interpret('.ATTACH LIB LIB1');
END;

EXCEPTION
/*
** Check for a specific error code, print the
** message, then discard the error from the stack
** If the error does not match, then raise it.
*/
WHEN OTHERS THEN
  IF Tool_Err.Code = Tool_Err.Encode('DEPLI',18) THEN
    Text_IO.Put_Line(Tool_Err.Message);
    Tool_Err.Pop;
  ELSE
    RAISE;
  END IF;
END;

If the exception handling code did not make use of Tool_Err constructs, you would have received an error alert displaying the message PDE-PLI018: Could not find library LIB1. Using Tool_Err constructs, the error is caught and the message is sent to the Interpreter.

### About the Tool_Res package

The Tool_Res package provides you with a means of extracting string resources from a resource file. The goal is to ease porting of PL/SQL code from one language to another by isolating all of the Textual Data in the resource file.

### Building resource files

In addition to extracting Textual Data from existing resource files, you can use the following utilities to create resource files that contain Textual Data.

- **RESPA21**
  Is a utility that generates a resource file (.RES) from a Text File (.PRN). The resulting resource file can be used with the Tool_Res Package.

- **RESPR21**
  Is a utility that converts a resource file (.RES) to a Text File (.PRN).

These utilities are distributed with Oracle*Terminal and are installed automatically with this product. To display the supported command line syntax of these utilities on your platform, run the utilities without supplying any arguments.
In Microsoft Windows, you can invoke these executables from the Explorer or File Manager to display their command line syntax. To run the executables with arguments, use Run.

**Resource File Syntax** Use the following syntax when you create strings for the resource file:

```plaintext
Resource resource_name
Type "string"
Content
table
  {  
    string string 1 character_count
      "content of string"
  }
```

where:

- `resource_name` is a unique name that you can reference with `Tool_Res.Rfread`.
- `character_count` is the number of characters in the string contents.
- `content of string` is the actual string.

**Example** The following Text file, HELLO.PRN:

```plaintext
Resource "hello_world"
Type "string"
Content
table
  {  
    string string 1 12
      "Hello World!"
  }

Resource "goodbye_world"
Type "string"
Content
table
  {  
    string string 1 14
      "Goodbye World!"
  }
```

is generated into the resource file HELLO.RES using the RESPA21 utility, and referenced by the following program unit:

```plaintext
PROCEDURE get_res IS
  resfileh Tool_Res.Rfhandle;
  hellor VARCHAR2(16);
  goodbyer VARCHAR2(16);
BEGIN
  /*Open the resource file we generated */
  resfileh:=Tool_Res.Rfopen('hello.res');

  /*Get the resource file strings*/
  hellor:=Tool_Res.Rfread(resfileh, 'hello_world');
  goodbyer:=Tool_Res.Rfread(resfileh, 'goodbye_world');

  /*Close the resource file*/

```

Forms Developer and Reports Developer Built-in Package Reference
About the EXEC_SQL package

The EXEC_SQL package allows you to access multiple Oracle database servers on several different connections at the same time. Connections can also be made to ODBC data sources via the Open Client Adapter (OCA), which is supplied with Forms Developer. To access non-Oracle data sources, you must install OCA and an appropriate ODBC driver.

The EXEC_SQL package contains procedures and functions you can use to execute dynamic SQL within PL/SQL procedures. Like the DBMS_SQL package, the SQL statements are stored in character strings that are only passed to or built by your source program at runtime. You can issue any data manipulation language (DML) or data definition language (DDL) statement using the EXEC_SQL package.

The EXEC_SQL package differs from the DBMS_SQL package in the following ways:

- Uses bind by value instead of bind by address
- Must use EXEC_SQL.Variable_Value to retrieve the value of an OUT bind parameter
- Must use EXEC_SQL.Column_Value after fetching rows to retrieve the values in a result set
- Does not support CHAR, RAW, LONG or ROWID data
- Does not provide a CANCEL_CURSOR procedure or function
- Does not support the array interface
- Indicator variables are not required because nulls are fully supported as values of PL/SQL variables
- Does not support PL/SQL tables or record types

For more information about the DBMS_SQL package, see your Oracle7 Application Developer's Guide or Oracle8 Application Developer's Guide.
Connection and cursor handles

In a Forms Developer application, you can have several connections to one or more databases at the same time. However, there is always one primary database connection, which we refer to as the primary Forms Developer connection. Handles are used to reference the Oracle or ODBC connections in your Forms Developer application. When you open connections to the primary database or to other databases, connection handles of type EXEC_SQL.ConnType are created and used to reference the connections. Each connection handle refers to one database connection. When you open a cursor on a connection handle, cursor handles of type EXEC_SQL.CursType are created and used to reference the cursor on the given connection. Each connection handle can have many cursor handles.

Data can be accessed after a connection and a cursor are opened. If you have multiple connections simultaneously opened, it is recommended that you explicitly include the specific handles as arguments in your EXEC_SQL routines. If you are only accessing data from the primary Forms Developer connection, then you do not need to specify the connection in the EXEC_SQL routines. When no handle is supplied to the EXEC_SQL routine, EXEC_SQL.Default_Connection is automatically called to obtain the primary Forms Developer connection.

Retrieving result sets from queries or non-Oracle stored procedures

The EXEC_SQL package is particularly useful when you need to retrieve result sets from different Oracle or ODBC data sources into one form or report.

To process a statement that returns a result set:

1. For each column, use EXEC_SQL.Define_Column to specify the variable for receiving the value.
2. Execute the statement by calling EXEC_SQL.Execute.
3. Use EXEC_SQL.Fetch_Rows to retrieve a row in the result set.
4. Use EXEC_SQL.Column_Value to obtain the value of each column retrieved by EXEC_SQL.Fetch_Rows.
5. Repeat 3 and 4 until EXEC_SQL.Fetch_Rows returns 0.
## EXEC_SQL predefined exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC_SQL.Invalid_Connection</td>
<td>An invalid connection handle is passed.</td>
</tr>
<tr>
<td>EXEC_SQL.Package_Error</td>
<td>Any general error. Use EXEC_SQL.Last_Error_Code and EXEC_SQL.Last_Error_Mesg to retrieve the error.</td>
</tr>
<tr>
<td>EXEC_SQL.Invalid_Column_Number</td>
<td>The EXEC_SQL.Describe_Column procedure encountered a column number that does not exist in the result set.</td>
</tr>
<tr>
<td>EXEC_SQL.Value_Error</td>
<td>The EXEC_SQL.Column_Value encountered a value that is different from the original value retrieved by EXEC_SQL.Define_Column.</td>
</tr>
</tbody>
</table>
Using the EXEC_SQL package

Executing arbitrary SQL against any connection
Copying data between two databases
Executing a non-Oracle database stored procedure and fetching its result set

Executing arbitrary SQL against any connection

The following procedure passes a SQL statement and an optional connection string of the form `user[/password]@[data source]`. If a connection string is passed, the procedure executes the SQL statement against the data source, otherwise it implements it against the primary Forms Developer connection.

```
PROCEDURE exec (sql_string IN VARCHAR2,
                connection_string IN VARCHAR2 DEFAULT NULL)
IS
    connection_id   EXEC_SQL.ConnType;
    cursor_number   EXEC_SQL.CursType;
    ret            PLS_INTEGER;
BEGIN
    IF connection_string IS NULL THEN
        connection_id := EXEC_SQL.DEFAULT_CONNECTION;
    ELSE
        connection_id := EXEC_SQL.OPEN_CONNECTION(connection_string);
    END IF;
    cursor_number := EXEC_SQL.OPEN_CURSOR(connection_id);
    EXEC_SQL.PARSE(connection_id, cursor_number, sql_string);
    ret := EXEC_SQL.EXECUTE(connection_id, cursor_number);
    EXEC_SQL.CLOSE_CURSOR(connection_id, cursor_number);
    EXEC_SQL.CLOSE_CONNECTION(connection_id);
```

-- Open a new connection. If the connection string is empty, assume the user wants to use the primary Forms Developer connection.
-- Open a cursor on the connection for executing the SQL statement.
-- Parse the SQL statement on the given connection.
-- And execute it. If the connection is Oracle, any DDL is done at parse time, but if the connection is a non-Oracle data source, this is not guaranteed.
-- Close the cursor.
-- And we are done with the connection. The connection_id we have may come from calling EXEC_SQL.OPEN_CONNECTION or EXEC_SQL.DEFAULT_CONNECTION.
EXCEPTION
  WHEN EXEC_SQL.PACKAGE_ERROR THEN
    TEXT_IO.PUT_LINE('ERROR (' ||
      TO_CHAR(EXEC_SQL.LAST_ERROR_CODE
      (connection_id)) || '): ' ||
      EXEC_SQL.LAST_ERROR_MESG(connection_id));
    IF EXEC_SQL.IS_CONNECTED(connection_id) THEN
      IF EXEC_SQL.IS_OPEN(connection_id, cursor_number) THEN
        EXEC_SQL.CLOSE_CURSOR(connection_id, cursor_number);
      END IF;
      EXEC_SQL.CLOSE_CONNECTION(connection_id);
    END IF;
END;

Regardless, we should call EXEC_SQL.CLOSE_CONNECTION. If the connection_id was obtained by EXEC_SQL.OPEN_CONNECTION,
EXEC_SQL.CLOSE_CONNECTION will terminate that connection. If the connection_id was obtained by EXEC_SQL.DEFAULT_CONNECTION,
EXEC_SQL.CLOSE_CONNECTION will NOT terminate that connection, but it frees up EXEC_SQL package specific resources.

-- This is the general error raised by the EXEC_SQL package, and denotes an unexpected error in one of the calls. It prints the error number and error message to standard out.
Copying data between two databases

The following procedure does not specifically require the use of dynamic SQL, but it illustrates the concepts in the EXEC_SQL package.

The procedure copies the rows from the source table (on the source connection) to the destination table (on the destination connection). It assumes the source and destination tables have the following columns:

| ID of type NUMBER |
| NAME of type VARCHAR2(30) |
| BIRTHDATE of type DATE |

PROCEDURE copy (source_table IN VARCHAR2,
source_connection IN VARCHAR2 DEFAULT NULL,
destination_connection IN VARCHAR2 DEFAULT NULL)
IS
    id NUMBER;
    name VARCHAR2(30);
    birthdate DATE;
    source_connid EXEC_SQL.ConnType;
    destination_connid EXEC_SQL.ConnType;
    source_cursor EXEC_SQL.CursType;
    destination_cursor EXEC_SQL.CursType;
    ignore PLS_INTEGER;
BEGIN
    IF source_connection IS NULL THEN
        source_connid := EXEC_SQL.DEFAULT_CONNECTION;
    ELSE
        source_connid := EXEC_SQL.OPEN_CONNECTION(source_connection);
    END IF;
    IF destination_connection IS NULL THEN
        destination_connid := EXEC_SQL.CURR_CONNECTION;
    ELSE
        destination_connid := EXEC_SQL.OPEN_CONNECTION(destination_connection);
    END IF;
    source_cursor := EXEC_SQL.OPEN_CURSOR(source_connid);
    EXEC_SQL.PARSE(source_connid, source_cursor, 'SELECT id, name, birthdate FROM ' || source_table);
    EXEC_SQL.DEFINE_COLUMN(source_connid, source_cursor, 1, id);
    EXEC_SQL.DEFINE_COLUMN(source_connid, source_cursor, 2, name, 30);
    EXEC_SQL.DEFINE_COLUMN(source_connid, source_cursor, 3, birthdate);
    ignore := EXEC_SQL.EXECUTE(source_connid, source_cursor);
    destination_cursor := EXEC_SQL.OPEN_CURSOR(destination_connid);
    EXEC_SQL.PARSE(destination_connid, destination_cursor, 'SELECT * FROM ' || destination_table);
    EXEC_SQL.DEFINE_COLUMN(destination_connid, destination_cursor, 1, id);
    EXEC_SQL.DEFINE_COLUMN(destination_connid, destination_cursor, 2, name, 30);
    EXEC_SQL.DEFINE_COLUMN(destination_connid, destination_cursor, 3, birthdate);
    ignore := EXEC_SQL.EXECUTE(destination_connid, destination_cursor);
'INSERT INTO ' || destination_table || '
(id, name, birthdate) VALUES (:id, :name, :birthdate)';

LOOP
IF EXEC_SQL.FETCH_ROWS(source_connid, source_cursor) > 0 THEN
    EXEC_SQL.COLUMN_VALUE(source_connid, source_cursor, 1, id);
    EXEC_SQL.COLUMN_VALUE(source_connid, source_cursor, 2, name);
    EXEC_SQL.COLUMN_VALUE(source_connid, source_cursor, 3, birthdate);
    EXEC_SQL.BIND_VARIABLE(destination_connid, destination_cursor, ':id', id);
    EXEC_SQL.BIND_VARIABLE(destination_connid, destination_cursor, ':name', name);
    EXEC_SQL.BIND_VARIABLE(destination_connid, destination_cursor, ':birthdate', birthdate);
    ignore := EXEC_SQL.EXECUTE(destination_connid, destination_cursor);
ELSE
    EXIT;
END LOOP;
EXEC_SQL.PARSE(destination_connid, destination_cursor, 'commit');
ignore := EXEC_SQL.EXECUTE(destination_connid, destination_cursor);
EXEC_SQL.CLOSE_CURSOR(destination_connid, destination_cursor);
EXEC_SQL.CLOSE_CURSOR(source_connid, source_cursor);
EXEC_SQL.CLOSE_CONNECTION(destination_connid);
EXEC_SQL.CLOSE_CONNECTION(source_connid);
EXCEPTION
WHEN EXEC_SQL.PACKAGE_ERROR THEN
    IF EXEC_SQL.LAST_ERROR_CODE(source_connid) != 0 THEN
        TEXT_IO.PUT_LINE('ERROR (source: ' || TO_CHAR(EXEC_SQL.LAST_ERROR_CODE(source_connid)) || '): ' || EXEC_SQL.LAST_ERROR_MESG(source_connid));
    END IF;
    IF EXEC_SQL.LAST_ERROR_CODE(destination_connid) != 0 THEN
        TEXT_IO.PUT_LINE('ERROR (destination: ' || TO_CHAR(EXEC_SQL.LAST_ERROR_CODE(destination_connid)) || '): ' || EXEC_SQL.LAST_ERROR_MESG(destination_connid));
    END IF;
    IF EXEC_SQL.IS_CONNECTED(destination_connid) THEN
        IF EXEC_SQL.IS_OPEN(destination_connid, destination_cursor) THEN
            EXEC_SQL.CLOSE_CURSOR(destination_connid, destination_cursor);
        END IF;
    END IF;
    IF EXEC_SQL.IS_CONNECTED(source_connid) THEN
        IF EXEC_SQL.IS_OPEN(source_connid, source_cursor) THEN
            EXEC_SQL.CLOSE_CURSOR(source_connid, source_cursor);
        END IF;
    END IF;
END IF;
EXEC_SQL.CLOSE_CONNECTION(destination_connid);
EXEC_SQL.CLOSE_CONNECTION(source_connid);

-- Fetch a row from the source table, and insert it into the destination table.
-- Get column values for the row; these are stored as local variables.
-- Bind the values into the cursor that inserts into the destination table.
-- No more rows to copy.
-- Commit the destination cursor.
-- And close everything.
-- This is the general error raised by the EXEC_SQL package. Get information (error number and message) about the error on the source connection or the destination connection.
-- Close all connections and cursors.
END IF;
END;


Executing a non-Oracle database stored procedure and fetching its result set

The following procedure executes a Microsoft SQL Server stored procedure that returns a result set, then fetches the result set. The stored procedure is:

```sql
create proc example_proc @id integer as
select ename from emp where empno = @id
```

The procedure executes the stored procedure on the primary Forms Developer connection (assuming it is a SQL Server connection), and prints out all lines returned. It is also assumed that the primary Forms Developer connection is already opened before executing procedure example3; otherwise an error will occur.

```sql
CREATE PROCEDURE example3 (v_id IN NUMBER) IS
  v_ename VARCHAR2(20);
  v_cur EXEC_SQL.CursType;
  v_rows INTEGER;
BEGIN
  v_cur := EXEC_SQL.OPEN_CURSOR;
  EXEC_SQL.PARSE(v_cur, '{ call example_proc ( :v_id ) }');
  EXEC_SQL.BIND_VARIABLE(v_cur, ':v_id', v_id);
  EXEC_SQL.DEFINE_COLUMN(v_curs, 1, v_ename, 20);
  v_rows := EXEC_SQL.EXECUTE(v_curs);
  WHILE EXEC_SQL.FETCH_ROWS(v_curs) > 0 LOOP
    EXEC_SQL.COLUMN_VALUE(v_curs, 1, v_ename);
    TEXT_IO.PUT_LINE('Ename = ' || v_ename);
  END LOOP;
  EXEC_SQL.CLOSE_CURSOR(v_cur);
EXCEPTION
  WHEN EXEC_SQL.PACKAGE_ERROR THEN
    TEXT_IO.PUT_LINE('ERROR (' || TO_CHAR(EXEC_SQL.LAST_ERROR_CODE) || '): ' || EXEC_SQL.LAST_ERROR_MESG);
    EXEC_SQL.CLOSE_CURSOR(v_cur);
  WHEN EXEC_SQL.INVALID_CONNECTION THEN
    TEXT_IO.PUT_LINE('ERROR: Not currently connected to a database');
END example3;
```

-- When no connection handle is passed, EXEC_SQL uses the primary Forms Developer connection.

-- To call stored procedures against ODBC datasources, use ODBC syntax, but parameters should be specified as Oracle parameters.

-- The exception INVALID_CONNECTION is raised when there is no default connection.
Alphabetic list of packaged subprograms

DDE.App_Begin
DDE.App_End
DDE.App_Focus
DDE.DMLERR_Not_Supported
DDE.Execute
DDE.Getformatnum
DDE.Getformatstr
DDE.Initiate
DDE.IsSupported
DDE.Poke
DDE.Request
DDE.Terminate
Debug.Break
Debug.Getx
Debug.Interpret
Debug.Setx
Debug.Suspend
EXEC_SQL.Open_Connection
EXEC_SQL.Curr_Connection
EXEC_SQL.Default_Connection
EXEC_SQL.Open_Cursor
EXEC_SQL.Parse
EXEC_SQL.Describe_Column
EXEC_SQL.Bind_Variable
EXEC_SQL.Define_Column
EXEC_SQL.Execute
EXEC_SQL.Execute_And_Fetch
EXEC_SQL.Fetch_Rows
EXEC_SQL.More_Result_Sets
EXEC_SQL.Column_Value
EXEC_SQL.Variable_Value
EXEC_SQL.Is_Open
EXEC_SQL.Close_Cursor
EXEC_SQL.Is_Connected
EXEC_SQL.Is_OCA_Connection
EXEC_SQL.Close_Connection
EXEC_SQL.Last_Error_Position
EXEC_SQL.Last_Row_Count
EXEC_SQL.Last_SQL_Function_Code
EXEC_SQL.Last_Error_Code
EXEC_SQL.Last_Error_Mesg
List.Appenditem
List.Destroy
List.Deleteitem
List.Fail
List.GetItem
List.Insertitem
List.Listofchar
List.Make
List.Nitems
List.Prependitem
OLE2.Add_Arg
OLE2.Create_Arglist
OLE2.Destroy_Arglist
OLE2.Get_Char_Property
OLE2.Get_Num_Property
OLE2.Get_Obj_Property
OLE2_Invoke
OLE2_Invoke_Num
OLE2_Invoke_Char
OLE2_Invoke_Obj
OLE2.IsSupported
OLE2.List_Type
OLE2.Obj_Type
OLE2.OLE_Not_Supported
OLE2_Release_Obj
OLE2.Set_Property
Ora_FFI.Find_Function
Ora_FFI.Find_Library
Ora_FFI.Funchandletype
Ora_FFI.Generate_Foreign
Ora_FFI.Is_Null_Ptr
Ora_FFI.Libhandletype
Ora_FFI.Load_Library
Ora_FFI_Pointertype
Ora_FFI_Register_Function
Ora_FFI_Register_Parameter
Ora_FFI_Register_Return
Ora_NLS_American
Ora_NLS_American_Date
Ora_NLS_Bad_Attribute
Ora_NLS_Get_Lang_Scalar
Ora_NLS.Get.Lang_Str
Ora_NLS.Linguistic_Collate
Ora_NLS.Linguistic_Specials
Ora_NLS.Modified_DateFmt
Ora_NLS.No_Item
Ora_NLS.Not_Found
Ora_NLS.Right_to_Left
Ora_NLS.Simple_CS
Ora_NLS.Single_Bray
Ora_Prof.Bad_Timer
Ora_Prof.Create_Timer
Ora_Prof.Destroy_Timer
Ora_Prof.Elapsed_Time
Ora_Prof.Reset_Timer
Ora_Prof.Start_Timer
Ora_Prof.Stop_Timer
Text_IO.FClose
Text_IO.File_Type
Text_IO.Fopen
Text_IO.Is_Open
Text_IO.Get_Line
Text_IO.New_Line
Text_IO.Put
Text_IO.PutF
Text_IO.Put_Line
Tool_Env.Getvar
Tool_Err.Clear
Tool_Err.Code
Tool_Err.Encode
Tool_Err.Message
Tool_Err.Nerrors
Tool_Err.Pop
Tool_Err.Tool_Error
Tool_Err.Toperror
Tool_Res.Bad_File_Handle
Tool_Res.Buffer_Overflow
Tool_Res.File_Not_Found
Tool_Res.No_Resource
Tool_Res.Rfclose
Tool_Res.Rfhandle
Tool_Res.Rfopen
Tool_Res.Rfread
DDE Package

DDE package
DDE.App_Begin
DDE.App_End
DDE.App_Focus
DDE.DMLERR_Not Supported
DDE.Execute
DDE.Getformatnum
DDE.Getformatstr
DDE.Initiate
DDE.IsSupported
DDE.Poke
DDE.Request
DDE.Terminate

DDE.App_Begin

**Description**  Begins an application program and returns an application identifier.

**Syntax**  
```
FUNCTION DDE.App_Begin
    (AppName  VARCHAR2,
    AppMode  PLS_INTEGER)
RETURN PLS_INTEGER;
```

**Parameters**
- **AppName**  The application name.
- **AppMode**  The application starting modes are:
  - **App_Mode_Normal**  Start the application window in normal size.
  - **App_Mode_Minimized**  Start the application window in minimized size.
  - **App_Mode_Maximized**  Start the application
window in maximized size.

**Returns**  An application identifier.

**Usage Notes**  The application name may contain a path. If the application name does not contain a path, then the following directories are searched in the order shown below:
- current directory
- Windows directory
- Windows system directory
- directory containing the executable file for the current task

For *AppName*, the application program name may be followed by arguments, which should be separated from the application program name with a space.

The application may be started in either normal, minimized, or maximized size, as specified by *AppMode*.

The application identifier returned by DDE.App_Begin must be used in all subsequent calls to DDE.App_End and DDE.App_Focus for that application window.

### DDE.App_Begin example

```
/*
** Start MS Excel with spreadsheet emp.xls loaded
*/
DECLARE
  AppID  PLS_INTEGER;
BEGIN
  AppID := DDE.App_Begin('c:\excel\excel.exe emp.xls',
                         DDE.App_Mode_Minimized);
END;
```

---

### DDE.App_End

**Description**  Ends an application program started by Dde_App_Begin.

**Syntax**

```
PROCEDURE DDE.App_End
    (AppID  PLS_INTEGER);
```

**Parameters**

- **AppID**  The application identifier returned by DDE.App_Begin.

**Usage Notes**  The application may also be terminated in standard Windows fashion: for example, by double-clicking the Control menu.
You must have previously called DDE.App_Begin to start the application program in order to end it using DDE.App_End.

**DDE.App_End example**

```plaintext
/*
** Start Excel, perform some operations on the
** spreadsheet, then close the application.
*/
DECLARE
    AppID  PLS_INTEGER;
BEGIN
    AppID := DDE.App_Begin('c:\excel\excel.exe emp.xls',
                          DDE.App_Mode_Normal);
    ...
    DDE.App_End(AppID);
END;
```

**DDE.App_Focus**

**Description** Activates an application program started by DDE.App_Begin.

**Syntax**

```plaintext
PROCEDURE DDE.App_Focus
    (AppID  PLS_INTEGER);
```

**Parameters**

- **AppID** The application identifier returned by DDE.App_Begin.

**Usage Notes** The application may also be activated in standard Windows fashion: for example, by clicking within the application window. To activate an application program using DDE.App_Focus, you must have previously called DDE.App_Begin to start the application program.

**DDE.App_Focus example**

```plaintext
/*
** Start Excel, then activate the application window
*/
DECLARE
    AppID  PLS_INTEGER;
BEGIN
    AppID := DDE.App_Begin('c:\excel\excel.exe',
                          DDE.App_Mode_Maximized);
    DDE.App_Focus(AppID);
END;
```
DDE.Execute

**Description** Executes a command string that is acceptable to the receiving server application.

**Syntax**

```plsql
PROCEDURE DDE.Execute
    (ConvID   PLS_INTEGER,
     CmdStr   VARCHAR2,
     Timeout  PLS_INTEGER);
```

**Parameters**

- **ConvID** The DDE Conversation identifier returned by DDE.Initiate.
- **CmdStr** The command string to be executed by the server.
- **Timeout** The timeout duration, in milliseconds.

**Usage Notes** The value of `CmdStr` depends on what values are supported by the server application. `Timeout` specifies the maximum length of time, in milliseconds, that this routine waits for a response from the DDE server application. If you specify an invalid number (e.g., a negative number), then the default value of 1000 ms is used.

**DDE.Execute example**

```plsql
/*
** Initiate Excel, then perform a recalculation
*/
DECLARE
    ConvID  PLS_INTEGER;
BEGIN
    ConvID := DDE.Initiate('EXCEL', 'abc.xls');
    DDE.Execute(ConvID, '[calculate.now()]', 1000);
END;
```

DDE.Getformatnum

**Description** Translates or registers a specified data format name and returns the numeric representation of the data format string.

**Syntax**

```plsql
FUNCTION DDE.Getformatnum
    (DataFormatName  VARCHAR2)
RETURN PLS_INTEGER;
```

**Parameters**

- **DataFormatName** The data format name string.
**Usage Notes**  DDE.Getformatnum converts a data format from a string to a number. This number can be used in DDE.Poke and DDE.Request transactions to represent the DataFormat variable.

If the specified name has not been registered yet, then DDE.Getformatnum registers it and returns a unique format number. This is the only way to use a format in a DDE.Poke or DDE.Request transaction that is not one of the predefined formats.

**DDE.Getformatnum example**

```sql
/*
** Get predefined format number for "CF_TEXT" (should return CF_TEXT=1) then register a user-defined data format called "MY_FORMAT"
*/
DECLARE
    FormatNum    PLS_INTEGER;
    MyFormatNum  PLS_INTEGER;
BEGIN
    FormatNum := DDE.Getformatnum('CF_TEXT');
    MyFormatNum := DDE.Getformatnum('MY_FORMAT');
END;
```

**DDE.Getformatstr**

**Description**  Translates a data format number into a format name string.

**Syntax**

```sql
FUNCTION DDE.Getformatstr
    (DataFormatNum  PLS_INTEGER)
RETURN VARCHAR2;
```

**Parameters**

- **DataFormat-Num**  A data format number.

**Returns**  The string representation of the supplied data format number.

**Usage Notes**  DDE.Getformatstr returns a data format name if the data format number is valid. Valid format numbers include the predefined formats and any user-defined formats that were registered with DDE.Getformatnum.

**DDE.Getformatstr example**

```sql
/*
** Get a data format name (should return the string 'CF_TEXT')
*/
DECLARE
```

---

**Forms Developer and Reports Developer Built-in Package Reference**  33
DDE.Initiate

**Description**  Opens a DDE conversation with a server application.

**Syntax**

```sql
FUNCTION DDE.Initiate
  (Service VARCHAR2,
   Topic VARCHAR2)
RETURN PLS_INTEGER;
```

**Parameters**

- **Service**  The server application’s DDE Service code.
- **Topic**  The topic name for the conversation.

**Returns**  A DDE Conversation identifier.

**Usage Notes**  The values of **Service** and **Topic** depend on the values supported by a particular DDE server application. **Service** is usually the name of the application program. For applications that operate on file-based documents, **Topic** is usually the document filename; in addition, the System topic is usually supported by each service. The conversation identifier returned by DDE.Initiate must be used in all subsequent calls to DDE.Execute, DDE.Poke, DDE.Request, and DDE.Terminate for that conversation. An application may start more than one conversation at a time with multiple services and topics, provided that the conversation identifiers are not interchanged. Use DDE.Terminate to terminate the conversation.

**DDE.Initiate example**

```sql
/*
** Open a DDE Conversation with MS Excel on
** topic abc.xls
*/
DECLARE
  ConvID  PLS_INTEGER;
BEGIN
  ConvID := DDE.Initiate('EXCEL', 'abc.xls');
END;
```
DDE.IsSupported

**Description**  Confirms that the DDE package is supported on the current platform.

**Syntax**  

DDE.ISSUPPORTED

**Returns**  TRUE, if DDE is supported on the platform; FALSE if it is not.

DDE.IsSupported example

```plsql
/*
** Before calling a DDE object in platform independent code,
** use this predicate to determine if DDE is supported on the
** current platform.
*/
IF (DDE.ISSUPPORTED) THEN
  . . . PL/SQL code using the DDE package
ELSE
  . . . message that DDE is not supported
END IF;
```

DDE.DMLERR_Not_Supported

**Description**  This exception is raised if a call is made to the DDE package but DDE is not supported on the current software platform.

**Syntax**  

DDE.DMLERR_NOT_SUPPORTED EXCEPTION;

DDE.Poke

**Description**  Sends data to a server application.

**Syntax**  

PROCEDURE DDE.Poke
  (ConvID      PLS_INTEGER,
   Item        VARCHAR2,
   Data        VARCHAR2,
   DataFormat  PLS_INTEGER,
   Timeout     PLS_INTEGER);

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConvID</td>
<td>The DDE Conversation identifier returned by DDE.Initiate.</td>
</tr>
<tr>
<td>Item</td>
<td>The data item name to which the data is to be sent.</td>
</tr>
</tbody>
</table>

Forms Developer and Reports Developer Built-in Package Reference 35
**Usage Notes** The value of *Item* depends on what values are supported by the server application on the current conversation topic. The predefined data format constants may be used for *DataFormat*. A user-defined format that was registered with DDE.GetFormatNum may also be used, provided that the server application recognizes this format. The user is responsible for ensuring that the server application will process the specified data format. *Timeout* specifies the maximum length of time, in milliseconds, that this routine waits for a response from the DDE server application. If you specify an invalid number (e.g., a negative number), then the default value of 1000 ms is used.

**DDE.Poke example**

```plaintext
/*
** Open a DDE Conversation with MS Excel on topic
** abc.xls and end data "foo" to cell at row 2,
** column 2
*/
DECLARE
  ConvID  PLS_INTEGER;
BEGIN
  ConvID = DDE.Initiate('EXCEL', 'abc.xls');
  DDE.Poke(ConvID, 'R2C2', 'foo', DDE.CF_TEXT, 1000);
END;
```

---

**DDE.Request**

**Description** Requests data from a server application.

**Syntax**

```plaintext
PROCEDURE DDE.Request
  (ConvID    PLS_INTEGER,
   Item      VARCHAR2,
   Buffer    VARCHAR2,
   DataFormat PLS_INTEGER,
   Timeout   PLS_INTEGER);
```

**Parameters**

- **ConvID**: The DDE Conversation identifier returned by DDE.Initiate.
- **Item**: Is requested data item name.
- **Buffer**: The result data buffer.
- **DataFormat**: The format of the requested buffer.
- **Timeout**: The timeout duration in milliseconds.
Usage Notes The value of `Item` depends on what values are supported by the server application on the current conversation topic. The user is responsible for ensuring that the return data buffer is large enough for the requested data. If the buffer size is smaller than the requested data, the data is truncated. The predefined data format constants may be used for `DataFormat`. A user-defined format that was registered with DDE.Getformatnum may also be used, provided that the server application recognizes this format. It is the user’s responsibility to ensure that the server application will process the specified data format. `Timeout` specifies the maximum length of time, in milliseconds, that this routine waits for a response from the DDE Server application. If the user specifies an invalid number, such as negative number, then the default value of 1000 ms is used.

DDE.Request example

```sql
/*
** Open a DDE Conversation with MS Excel for Windows on
** topic abc.xls then request data from 6 cells
** between row 2, column 2 and row 3, column 4
*/
DECLARE
    ConvID   PLS_INTEGER;
    Buffer   VARCHAR2(80);
BEGIN
    ConvID := DDE.Initiate('EXCEL', 'abc.xls');
    DDE.Request (ConvID, 'R2C2:R3C4', Buffer, DDE.Cf_Text, 1000);
END;
```

DDE.Terminate

Description Terminates the specified conversation with an application.

Syntax

```sql
PROCEDURE DDE.Terminate
    (ConvID  PLS_INTEGER);
```

Parameters

- **ConvID** The conversation identifier.

Usage Notes After the DDE.Terminate call, all subsequent calls to DDE.Execute, DDE.Poke, DDE.Request, and DDE.Terminate using the terminated conversation identifier will result in an error.

To terminate a conversation with a server application using DDE.Terminate, you must have used DDE.Initiate to start the conversation.
DDE.Terminate example

/*
** Open a DDE Conversation with MS Excel on topic 
** abc.xls perform some operations, then terminate  
** the conversation 
*/
DECLARE  
    ConvID    PLS_INTEGER;
BEGIN  
    ConvID := DDE.Initiate('EXCEL', 'abc.xls'); 
    ...  
    DDE.Terminate(ConvID); 
END;
Debug Package

Debug package
Debug.Break
Debug.Getx
Debug.Interpret
Debug.Setx
Debug.Suspend

Debug.Break

Description Used to enter a breakpoint from within a debug trigger.
Syntax
Debug.Break EXCEPTION;
Usage Notes Debug.Break is very useful for creating conditional breakpoints. When the exception is raised, control is passed to the Interpreter as if you had entered a breakpoint at the debug trigger location.

Debug.Break example

/*
** Create a breakpoint only when the value
** of 'my_sal' exceeds 5000
*/
IF Debug.Getn('my_sal') > 5000 THEN
   RAISE Debug.Break;
END IF;

Debug.Getx

Description Retrieve the value of the specified local variable.
Syntax

```sql
FUNCTION Debug.Getc
    (varname VARCHAR2)
RETURN VARCHAR2;
FUNCTION Debug.Getd
    (varname VARCHAR2)
RETURN DATE;
FUNCTION Debug.Geti
    (varname VARCHAR2)
RETURN PLS_INTEGER;
FUNCTION Debug.Getn
    (varname VARCHAR2)
RETURN NUMBER;
```

Parameters

<table>
<thead>
<tr>
<th>varname</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A VARCHAR2 or CHAR (Debug.Getc converts CHAR values to VARCHAR2), DATE, PLS_INTEGER, or NUMBER variable.</td>
<td></td>
</tr>
</tbody>
</table>

Usage Notes

This is useful when you want to determine a local’s value from within a debug trigger.

Debug.Getx examples

```sql
/*
** Retrieve the value of the variable ‘my_ename’
** and use it to test a condition
*/
IF Debug.Getc('my_ename') = 'JONES' THEN
    RAISE Debug.Break;
END IF;
```

You have a program unit `foo` that calls the subprogram `bar`. That subprogram (`bar`) is also called by many other program units. Consider the situation where procedure `bar` accepts the argument `message` from the many procedures that call it. Procedure `foo` passes a unique argument of ‘hello world’ to `bar`. In this case, we could define a trigger that raises a breakpoint in procedure `bar` only when `foo` passes its argument:

```sql
PL/SQL> .TRIGGER PROC bar LINE 3 IS
    >BEGIN
    >  IF Debug.Getn!(‘message’) = ‘hello world’ THEN
    >    RAISE Debug.Break;
    >  END IF;
    >END;
```
Debug.Interpret

**Description**  Executes the PL/SQL statement or Procedure Builder Interpreter command string contained in input as if it had been typed into the Interpreter.

**Syntax**

PROCEDURE Debug.Interpret
  (input VARCHAR2);

**Parameters**

input  A Procedure Builder command string.

**Usage Notes**  This is useful for automatically invoking Procedure Builder functions from a debug trigger.

**Debug.Interpret examples**

```sql
/*
** Execute the command SHOW STACK when
** a condition is met
*/
IF Debug.Getc('my_ename') = 'JONES' THEN
  Debug.Interpret('.SHOW LOCALS');
END IF;
```

You have a program unit foo that calls the subprogram bar. That subprogram (bar) is also called by many other program units. You want to create a breakpoint in bar, but you only want to enable the breakpoint when the subprogram is called from foo and not when it is called from other program units.

To do this, you need to perform the following steps:

1. Create a breakpoint in procedure bar where you want to suspend execution.
2. Disable the breakpoint you just created.

You can perform both steps 1 and 2 from within the Breakpoint dialog box. Create a breakpoint with a breakpoint trigger in procedure foo that enables the first breakpoint we created in procedure bar. For example:

```sql
PL/SQL>.BREAK PROC foo LINE 6 TRIGGER
  >BEGIN
  >  Debug.Interpret('.enable break 1');
  >  Debug.Interpret('.go');
  >END;
```

The following example creates a breakpoint which fires a trigger each time the breakpoint is hit.

```sql
PL/SQL> .break proc my_proc line 10 trigger
  => DEBUG.INTERPRET('.SHOW LOCALS');
```
Debug.Setx

Description  Set the value of a local variable to a new value.

Syntax
PROCEDURE Debug.Setc(
  varname  VARCHAR2,
  newvalue VARCHAR2);
PROCEDURE Debug.Setd(
  varname  VARCHAR2,
  newvalue  DATE);
PROCEDURE Debug.Seti(
  varname   VARCHAR2,
  newvalue PLS_INTEGER);
PROCEDURE Debug.Setn(
  varname   VARCHAR2,
  newvalue NUMBER);

Parameters
  varname  A VARCHAR2 or CHAR (Debug.Setc converts CHAR values to VARCHAR2), DATE, PLS_INTEGER, or NUMBER variable.
  newvalue An appropriate value for varname.

Usage Notes  This is useful when you want to change a local’s value from a debug trigger.

Debug.Setx examples

  /*
   ** Set the value of the local variable ‘my_emp’ from a Debug Trigger
   */
  Debug.Setc(‘my_emp’, ‘SMITH’);
  /*
   ** Set the value of the local variable ‘my_date’ from a Debug Trigger
   */
  Debug.Setd(‘my_date’, ‘02-OCT-94’);

Debug.Suspend

Description  Suspends execution of the current program unit and transfers control to the Interpreter.

Syntax
PROCEDURE Debug.Suspend;
Debug.Suspend example

/*
** This example uses Debug.Suspend
*/
PROCEDURE proc1 IS
BEGIN
 FOR i IN 1..10 LOOP
  Debug.Suspend;
  Text_IO.Put_Line('Hello');
 END LOOP;
END;
EXEC_SQL Package

EXEC_SQL package
The functions and procedures are listed in the order they are usually called in a session.
EXEC_SQL.Open_Connection
EXEC_SQL.Curr_Connection
EXEC_SQL.Default_Connection
EXEC_SQL.Open_Cursor
EXEC_SQL.Parse
EXEC_SQL.Describe_Column
EXEC_SQL.Bind_Variable
EXEC_SQL.Define_Column
EXEC_SQL.Execute
EXEC_SQL.Execute_And_Fetch
EXEC_SQL.Fetch_Rows
EXEC_SQL.More_Result_Sets
EXEC_SQL.Column_Value
EXEC_SQL.Variable_Value
EXEC_SQL.Is_Open
EXEC_SQL.Close_Cursor
EXEC_SQL.Is_Connected
EXEC_SQL.Is_OCA_Connection
EXEC_SQL.Close_Connection

The following functions retrieve information about the last referenced cursor in a
connection after a SQL statement execution.
EXEC_SQL.Last_Error_Position
EXEC_SQL.Last_Row_Count
EXEC_SQL.Last_SQL_Function_Code
EXEC_SQL.Last_Error_Code
EXEC_SQL.Last_Error_Mesg
EXEC_SQL.Open_Connection

FUNCTION EXEC_SQL.Open_Connection
(Username      IN VARCHAR2,
 Password      IN VARCHAR2,
 Data source   IN VARCHAR2)
RETURN EXEC_SQL.ConnType;

Parameters

Connstr      Is a string in the form
             'User[/Password][/@database_string]' 
Username      A string specifying the user name used to
              connect to the database
Password      A string specifying the password for the user
              name
Data source   Either a string specifying the SQLNet alias or
              the OCA connection starting with 'ODBC:'

Returns   A handle to the new database connection.

EXEC_SQL.Open_Connection example

PROCEDURE getData IS
  ...
  -- a connection handle must have a datatype of EXEC_SQL.conntype
  --
  connection_id EXEC_SQL.CONNTYPE;
  ...
BEGIN
  ...
  -- a connection string is typically of the form
  -- 'username/password@database_alias'
  --
  connection_id := EXEC_SQL.OPEN_CONNECTION('connection_string');
  ...
END;

EXEC_SQL.Curr_Connection

Description   Returns a connection handle that uses the same database connection
               originally established by Forms Developer. EXEC_SQL.Default_Connection replaces
               EXEC_SQL.Curr_Connection.

Syntax
FUNCTION EXEC_SQL.Curr_Connection
RETURN EXEC_SQL.ConnType;

Returns A handle to the primary Forms Developer connection.
Usage notes Use EXEC_SQL.Default_Connection in place of
EXEC_SQL.Curr_Connection. For backward compatibility,
EXEC_SQL.Curr_Connection is still supported.

EXEC_SQL.Default_Connection

Description Returns a connection handle that uses the same database connection
originally established by Forms Developer. EXEC_SQL.Default_Connection replaces
EXEC_SQL.Curr_Connection.
Syntax
FUNCTION EXEC_SQL.Default_Connection
RETURN EXEC_SQL.ConnType;

Returns A handle to the primary Forms Developer connection.
Usage notes The default connection is the primary Forms Developer connection. The
first time EXEC_SQL.Default_Connection is called, the default connection is found,
placed in a cache within the EXEC_SQL package, and a handle is returned to the user.
Subsequent calls to EXEC_SQL.Default_Connection simply retrieves the handle from the
cache.
Since this connection handle is cached, if you are accessing data from only the default
connection, then you do not need to explicitly specify the connection handle in calls to
other EXEC_SQL methods; EXEC_SQL automatically looks up the cache to obtain the
connection handle.
To clear the cache, call EXEC_SQL.Close_Connection on the connection handle that is
obtained from calling EXEC_SQL.Default_Connection. For default connections,
EXEC_SQL.Close_Connection does not terminate the connection, but only frees up the
resources used by EXEC_SQL.

EXEC_SQL.Default_Connection and
EXEC_SQL.Curr_Connection example

/**
 ** This example illustrates the use of
 ** EXEC_SQL.Default_Connection and
 ** EXEC_SQL.Curr_Connection.
 */
PROCEDURE esdefaultcon2 IS
   connection_id EXEC_SQL.CONNTYPE;
   bIsConnected BOOLEAN;
   cursorID EXEC_SQL.CURSTYPE;
   sqlstr VARCHAR2(1000);
   nIgn PLS_INTEGER;

   Forms Developer and Reports Developer Built-in Package Reference 47
nRows PLS_INTEGER := 0;
nTimes PLS_INTEGER := 0;
mynum NUMBER;
BEGIN
   --
   -- obtain the default connection and check that it is valid
   --
   connection_id := EXEC_SQL.DEFAULT_CONNECTION;
bIsConnected := EXEC_SQL.IS_CONNECTED;
   IF bIsConnected = FALSE THEN
      TEXT_IO.PUT_LINE('No primary connection. Please connect before
retrying.');
      RETURN;
   END IF;
   --
   -- subsequent calls to EXEC_SQL.Open_Cursor, EXEC_SQL.Parse,
EXEC_SQL.Define_Column,
   -- EXEC_SQL.Execute, EXEC_SQL.Fetch_Rows, EXEC_SQL.Column_Value,
   -- EXEC_SQL.Close_Cursor, EXEC_SQL.Close_Connection all use this
connection
   -- implicitly from the cache
   --
   cursorID := EXEC_SQL.OPEN_CURSOR;
   sqlstr := 'select empno from emp';
   EXEC_SQL.PARSE(cursorID, sqlstr, exec_sql.V7);
   EXEC_SQL.DEFINE_COLUMN(cursorID, 1, mynum);
   nIgn := EXEC_SQL.EXECUTE(cursorID);
   LOOP
      IF (EXEC_SQL.FETCH_ROWS(cursorID) > 0) THEN
         EXEC_SQL.COLUMN_VALUE(cursorID, 1, mynum);
          ...
      ELSE
         exit;
      END IF;
   END LOOP;
   EXEC_SQL.CLOSE_CURSOR(cursorID);
   EXEC_SQL.CLOSE_CONNECTION;
END;

**EXEC_SQL.Open_Cursor**

**Description**  Creates a new cursor on a specified connection and returns a cursor handle. When you no longer need the cursor, you must close it explicitly by using EXEC_SQL.Close_Cursor.

**Syntax**

FUNCTION EXEC_SQL.Open_Cursor
   [Connid IN CONNTYPE]
RETURN EXEC_SQL.CursType;
**Parameters**

*Connid*  
Is the handle to the connection you want to use.  
If you do not specify a connection,  
EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

**Returns**  
A handle to the new cursor.

**Usage Notes**  
You can use cursors to execute the same SQL statement repeatedly (without reparsing) or to execute a new SQL statement (with parsing). When you reuse a cursor for a new statement, the cursor contents are automatically reset when the new statement is parsed. This means you do not have to close and reopen a cursor before reusing it.

**EXEC_SQL.Open_Cursor example**

```plaintext
PROCEDURE getData IS
  --
  -- a cursorID must be of type EXEC_SQL.cursType
  --
  connection_id EXEC_SQL.CONNTYPE;
  cursorID EXEC_SQL.CURSTYPE;
  ...
BEGIN
  connection_id := EXEC_SQL.OPEN_CONNECTION('connect_str');
  ...
  --
  -- this cursor is now associated with a particular connection
  --
  cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
  ...
END;
```

**EXEC_SQL.Parse**

**Description**  
This procedure parses a statement on a specified cursor.

**Syntax**

```plaintext
PROCEDURE EXEC_SQL.Parse
  (Connid IN CONNTYPE,
   Curs_Id IN CURSTYPE,
   Statement IN VARCHAR2,
   [Language IN PLS_INTEGER]);
```

---

**Forms Developer and Reports Developer Built-in Package Reference**

---

49
Parameters

Connid  
Is the handle to the connection you want to use.
If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

Curs_Id  
Is the cursor handle you want to assign the statement to.

Statement  
The SQL statement to be parsed. It should not include a final semicolon.

Language_flag  
A flag that determines how Oracle handles the SQL statement. The valid flags are:

  - V6   Specifies Oracle V6 behavior
  - V7   Specifies Oracle V7 behavior
  - NATIVE   Default

Usage Notes  
All SQL statements must be parsed using the Parse procedure. Parsing checks the syntax of the statement and associates it with the cursor in your code. Unlike OCI parsing, EXEC_SQL parsing is always immediate. You cannot defer EXEC_SQL parsing.

You can parse any data manipulation language (DML) or data definition language (DDL) statement. For Oracle data sources, the DDL statements are executed on the parse. For non-Oracle data sources, the DDL may be executed on the parse or on the execute. This means you should always parse and execute all DDL statements in EXEC_SQL.

EXEC_SQL.Parse example

```sql
PROCEDURE getData IS
  connection_id EXEC_SQL.CONNTYPE;
  cursorID EXEC_SQL.CURSTYPE;
  sqlstr VARCHAR2(1000);
  ...  
  BEGIN
  connection_id := EXEC_SQL.OPEN_CONNECTION(connect_str);
  cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
  --
  -- the statement to be parsed is stored as a VARCHAR2 variable
  --
  sqlstr := 'select ename from emp';
  --
  -- perform parsing
  --
  EXEC_SQL.PARSE(connection_id, cursorID, sqlstr, exec_sql.V7);
  ...
END;
```

50   Forms Developer and Reports Developer Built-in Package Reference
EXEC_SQL.Describe_Column

Description Obtains information about the columns in a result set of a parsed SQL statement. If you try to describe a column number that does not exist in the result set, the EXEC_SQL.Invalid_Column_Number exception is raised. Tip

Syntax

PROCEDURE EXEC_SQL.Describe_Column
((Connid IN CONNTYPE,
  Curs_Id IN CURSTYPE,
  Position IN PLS_INTEGER,
  Name OUT VARCHAR2,
  Collen OUT PLS_INTEGER,
  Type OUT PLS_INTEGER));

Parameters

Connid Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

Curs_Id Is the cursor handle associated to the column you want to describe.

Position Is the position in the result set of the column you want to describe. The positions are numbered from left to right, starting at 1.

Name Contains the name of the column, on output.

Collen Contains the maximum length of the column in bytes, on output.

Type Contains the type of the column, on output. The valid values are one the following:
 EXEC_SQL.VARCHAR2_TYPE
 EXEC_SQL.NUMBER_TYPE
 EXEC_SQL.FLOAT_TYPE
 EXEC_SQL.LONG_TYPE
 EXEC_SQL.ROWID_TYPE
 EXEC_SQL.DATE_TYPE
 EXEC_SQL.RAW_TYPE
 EXEC_SQL.LONG_RAW_TYPE
 EXEC_SQL.CHAR_TYPE (ANSI fixed CHAR)
 EXEC_SQL.MLSLABLE_TYPE (Trusted Oracle only)
EXEC_SQL.Describe_Column example

PROCEDURE esdesccol(tablename VARCHAR2) IS
  connection_id EXEC_SQL.CONNTYPE;
  cursor_number EXEC_SQL.CURSTYPE;
  sql_str VARCHAR2(256);
  nIgnore PLS_INTEGER;
  nColumns PLS_INTEGER := 0;  --count of number of columns returned
  colName VARCHAR2(30);
  colLen PLS_INTEGER;
  colType PLS_INTEGER;

BEGIN
  connection_id := EXEC_SQL.OPEN_CONNECTION('connection_string');
  --
  -- when you do a "select *..." from a table which is known only at
  -- runtime,
  -- you cannot know what the columns are a priori.
  EXEC_SQL.Describe_Column becomes
  -- very usefule then
  --
  sql_str := 'select * from ' || tablename;
  cursor_number := EXEC_SQL.OPEN_CURSOR(connection_id);
  EXEC_SQL.PARSE(connection_id, cursor_number, sql_str, exec_sql.V7);
  nIgnore := EXEC_SQL.EXECUTE(connection_id, cursor_number);
  LOOP
    nColumns := nColumns + 1;  --used as column index into result set
    --
    -- describe_column is in general used within a PL/SQL block with an
    -- exception
    -- block included to catch the EXEC_SQL.invalid_column_number
    -- exception.
    -- when no more columns are found, we can store the returned column
    -- names
    -- and column lengths in a PL/SQL table of records and do further
    -- queries
    -- to obtain rows from the table. In this example, colName, colLen
    -- and colType
    -- are used to store the returned column characteristics.
    --
    BEGIN
      EXEC_SQL.DESCRIBE_COLUMN(connection_id, cursor_number,
        nColumns, colName, colLen, colType);
      TEXT_IO.PUT_LINE(' col=' || nColumns || ' name ' || colName
        || ' len=' || colLen || ' type ' || colType);
    EXCEPTION
      WHEN EXEC_SQL.INVALID_COLUMN_NUMBER THEN
        EXIT;
    END;
  END LOOP;
  nColumns := nColumns - 1;
  IF (nColumns <= 0) THEN
TEXT_IO.PUT_LINE('No columns returned in query');
END IF;
...
EXEC_SQL.CLOSE_CURSOR(connection_id, cursor_number);
EXEC_SQL.CLOSE_CONNECTION(connection_id);
END;

EXEC_SQL.Bind_Variable

Description  Binds a given value to a named variable in a SQL statement.

Syntax
PROCEDURE EXEC_SQL.Bind_Variable
(
Connid    IN CONNTYPE,  
Curs_Id   IN CURSTYPE,  
Name      IN VARCHAR2,  
Value     IN <datatype>);

where <datatype> can be one of the following:
NUMBER
DATE
VARCHAR2

PROCEDURE EXEC_SQL.Bind_Variable
(
Connid           IN CONNTYPE,  
Curs_Id          IN CURSTYPE,  
Name             IN VARCHAR2,  
Value            IN VARCHAR2,  
Out_Value_Size   IN PLS_INTEGER);

Parameters
Connid         Is the handle to the connection you want to use.
               If you do not specify a connection,
               EXEC_SQL.Default_Connection retrieves the
               primary Forms Developer connection handle
               from the cache.
Curs_Id        Is the cursor handle in which to bind the variable.
Name           Is the name of the variable in the SQL statement.
Value          For IN and IN/OUT variables, the value is the
data you want to bind to the named variable. For
OUT variables, the data is actually ignored but
you must still use Bind_Variable to indicate the
type of PL/SQL variable to be retrieved later by
Variable_Value.
Out_Value_Size  The maximum OUT value size in bytes expected
               for the VARCHAR2 OUT or IN/OUT variables.
               If no size is specified, the current length of the...
Usage Notes Use placeholders in SQL statements to mark where input data is to be supplied during runtime. You must also use placeholders for output values if the statement is a PL/SQL block or a call to a stored procedure with output parameters. For each input placeholder, you must use EXEC_SQL.Bind_Variable to supply the value. For each output placeholder, you must also use EXEC_SQL.Bind_Variable to specify the type of variable to use for retrieving the value in subsequent EXEC_SQL.Variable_Value calls.

The input placeholder or bind variable in a SQL statement is identified by a name beginning with a colon. For example, the string ':X' is the bind variable in the following SQL statement:

```
SELECT ename FROM emp WHERE SAL > :X;
```

The corresponding EXEC_SQL.Bind_Variable procedure is:

```sql
BIND_VARIABLE(connection_handle, cursor_handle, ':X', 3500);
```

EXEC_SQL.Bind_Variable example

```
PROCEDURE getData(input_empno NUMBER) IS
  connection_id EXEC_SQL.CONNTYPE;
  cursorID EXEC_SQL.CURSTYPE;
  sqlstr VARCHAR2(1000);
  ...

BEGIN
  connection_id := EXEC_SQL.OPEN_CONNECTION(connect_str);
  cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
  -- the statement to be parsed contains a bind variable
  sqlstr := 'select ename from emp where empno = :bn';
  -- perform parsing
  EXEC_SQL.PARSE(connection_id, cursorID, sqlstr, exec_sql.V7);
  -- the bind_variable procedure assigns the value of the input argument
  -- to the named
  -- bind variable. Note the use of the semi-colon and the quotes to designate the
  -- bind variable. The bind_variable procedure is called after the parse procedure.
  EXEC_SQL.BIND_VARIABLE(connection_id, cursorID, ':bn', input_empno);
  EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 1, mynum);
  ...
END;
```
EXEC_SQL.Define_Column

Description  This procedure is used only with SELECT statements or calls to non-Oracle stored procedures that return a result set. It defines a column to be fetched from a specified cursor. The column is identified by its relative position in the result set; the first relative position is identified by the integer 1. The PL/SQL type of the Column parameter determines the type of the column being defined.

Syntax

PROCEDURE EXEC_SQL.Define_Column
  ([Connid      IN CONNTYPE],
   Curs_Id     IN CURSTYPE,
   Position    IN PLS_INTEGER,
   Column      IN <datatype>);

where <datatype> can be one of the following:
   NUMBER
   DATE
   VARCHAR2

PROCEDURE EXEC_SQL.Define_Column
  ([Connid        IN CONNTYPE],
   Curs_Id       IN CURSTYPE,
   Position      IN PLS_INTEGER,
   Column        IN VARCHAR2,
   Column_Size   IN PLS_INTEGER);

Parameters

Connid           Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

Curs_Id          Is the cursor handle you want to define the column for.

Position         Is the relative position of the column in the row or result set. The first column in the statement has a relative position of 1.

Column           Is the value of the column being defined. The value type determines the column type being defined. The actual value stored in the variable is ignored.

Column_Size      Is the maximum expected size of the column value in bytes (for column type VARCHAR2 only)

Usage Notes    For a query, you must define the column before retrieving its data by EXEC_SQL.Column_Value.
EXEC_SQL.Define_Column example

PROCEDURE getData IS
    connection_id EXEC_SQL.CONNTYPE;
    cursorID EXEC_SQL.CURSTYPE;
    sqlstr VARCHAR2(1000);
    loc_ename VARCHAR2(30);   -- these are variables local to the
    procedure;
    loc_eno NUMBER;           -- used to store the return values from our
    desired
    loc_hiredate DATE;        -- query

    BEGIN
        connection_id := EXEC_SQL.OPEN_CONNECTION(connect_str);
        cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
        sqlstr := 'select ename, empno, hiredate from emp ';
        EXEC_SQL.PARSE(connection_id, cursorID, sqlstr, exec_sql.V7);
        EXEC_SQL.BIND_VARIABLE(connection_id, cursorID, ':bn', input_empno);
        -- we make one call to DEFINE_COLUMN per item in the select list. We
        must use local
        -- variables to store the returned values. For a result value that is
        a VARCHAR, it
        -- is important to specify the maximumn length. For a result value
        that is a number
        -- or a date, there is no need to specify the maximum length. We
        obtain the
        -- relative positions of the columns being returned from the select
        statement,
        -- sql_str.
        --
        EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 1, loc_ename, 30);
        EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 2, loc_eno);
        EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 3, loc_hiredate);

        ... END;

EXEC_SQL.Execute

Description   Executes the SQL statement at a specified cursor.

Syntax

FUNCTION EXEC_SQL.Execute
    ([Connid IN CONNTYPE],
     Curs_Id IN CURSTYPE)
RETURN PLS_INTEGER;

Parameters

Connid       Is the handle to the connection you want to use.
             If you do not specify a connection,
**EXEC_SQL.Default_Connection** retrieves the primary Forms Developer connection handle from the cache.

**Curs_Id**
Is the cursor handle to the SQL statement you want to execute.

**Returns**
The number of rows processed.

**Usage Notes**
The return value is only valid for INSERT, UPDATE and DELETE statements. For other statements, including DDL, ignore the return value because it is undefined.

**EXEC_SQL.Execute example**

```sql
PROCEDURE getData IS
    connection_id EXEC_SQL.CONNTYPE;
    cursorID EXEC_SQL.CURSTYPE;
    sqlstr VARCHAR2(1000);
    loc_ename VARCHAR2(30);
    loc_eno NUMBER;
    loc_hiredate DATE;
    nIgn PLS_INTEGER;

    BEGIN
        connection_id := EXEC_SQL.OPEN_CONNECTION(connect_str);
        cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
        sqlstr := 'select ename, empno, hiredate from emp ';
        EXEC_SQL.PARSE(connection_id, cursorID, sqlstr, exec_sql.V7);
        EXEC_SQL.BIND_VARIABLE(connection_id, cursorID, ':bn', input_empno);
        EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 1, loc_ename, 30);
        EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 2, loc_eno);
        EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 3, loc_hiredate);
        -- after parsing, and calling BIND_VARIABLE and DEFINE_COLUMN, if necessary, you
        -- are ready to execute the statement. Note that all information about the
        -- statement and its result set is encapsulated in the cursor
        -- referenced as cursorID.
        --
        nIgn := EXEC_SQL.EXECUTE(connection_id, cursorID);

    END;
```
EXEC_SQL.Execute_And_Fetch

**Description**  This function calls EXEC_SQL.Execute and then EXEC_SQL.Fetch_Rows. It executes a SQL statement at a specified cursor and retrieves the first row that satisfies the query. Calling EXEC_SQL.Execute_And_Fetch may reduce the number of round-trips when used against a remote database.

**Syntax**

```sql
FUNCTION EXEC_SQL.Execute_And_Fetch
    ([Connid IN CONNTYPE],
     Curs_Id IN CURSTYPE,
     Exact IN BOOLEAN DEFAULT FALSE)
RETURN PLS_INTEGER;
```

**Parameters**

- **Connid**
  Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

- **Curs_Id**
  Is the cursor handle to the SQL statement you want to execute.

- **Exact**
  The default is FALSE. Set to TRUE to raise the exception EXEC_SQL.Package_Error. The row is retrieved even if the exception is raised.

**Returns**  The number of rows fetched (either 0 or 1).

**EXEC_SQL.Execute_And_Fetch example**

```sql
PROCEDURE getData(input_empno NUMBER) IS
    connection_id EXEC_SQL.CONNTYPE;
    cursorID EXEC_SQL.CURSTYPE;
    sqlstr VARCHAR2(1000);
    loc_ename VARCHAR2(30);
    loc_eno NUMBER;
    loc_hiredate DATE;
    nIgn PLS_INTEGER;
    ...
    BEGIN
        connection_id := EXEC_SQL.OPEN_CONNECTION(connect_str);
        cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
        -- assuming that empno is a primary key of the table emp, the where clause guarantees
        -- that 0 or 1 row is returned
        --
        sqlstr := 'select ename, empno, hiredate from emp ' ||
                    'where empno = ' || input_empno;
```
EXEC_SQL.PARSE(connection_id, cursorID, sqlstr, exec_sql.V7);
EXEC_SQL.BIND_VARIABLE(connection_id, cursorID, ':bn', input_empno);
EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 1, loc_ename, 30);
EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 2, loc_eno);
EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 3, loc_hiredate);
--  do execute_and_fetch after parsing the statement, and calling
--  bind_variable and
--  define_column if necessary
--
nIgn := EXEC_SQL.EXECUTE_AND_FETCH (connection_id, cursorID);
IF (nIgn = 0 ) THEN
  TEXT_IO.PUT_LINE (' No employee has empno = ' || input_empno);
ELSE IF (nIgn = 1) THEN
  TEXT_IO.PUT_LINE (' Found one employee with empno ' || input_empno);
--  obtain the values in this row
--
  EXEC_SQL.column_value(connection_id, cursorID, 1, loc_ename);
  EXEC_SQL.column_value(connection_id, cursorID, 2, loc_eno);
  EXEC_SQL.column_value(connection_id, cursorID, 3, loc_hiredate);
...
END IF;
...
END;

---

EXEC_SQL.Fetch_Rows

Description  Retrieves a row that satisfies the query at a specified cursor.

Syntax
FUNCTION EXEC_SQL.Fetch_Rows
  ((Connid      IN CONNTYPE),
    Curs_Id     IN CURSTYPE)
RETURN PLS_INTEGER;

Parameters

Connid
  Is the handle to the connection you want to use.
  If you do not specify a connection,
  EXEC_SQL.Default_Connection retrieves the
  primary Forms Developer connection handle
  from the cache.

Curs_Id
  Is the cursor handle to the SQL statement from
  which you want to fetch.

Returns  The number of rows actually fetched.
**Usage Notes** Each EXEC_SQL.Fetch_Rows call retrieves one row into a buffer. Use EXEC_SQL.Fetch_Rows repeatedly until 0 is returned. For Oracle databases, this means there is no more data in the result set. For non-Oracle data sources, this does not mean there is no more data in the specified cursor. See EXEC_SQL.More_Results_Sets for more information.

After each EXEC_SQL.Fetch_Rows call, use EXEC_SQL.Column_Value to read each column in the fetched row.

**EXEC_SQL.Fetch_Rows example**

```sql
PROCEDURE getData IS
    connection_id EXEC_SQL.CONNTYPE;
    cursorID EXEC_SQL.CURSTYPE;
    sqlstr VARCHAR2(1000);
    loc_ename VARCHAR2(30);
    loc_eno NUMBER;
    loc_hiredate DATE;
    nIgn PLS_INTEGER;
    nRows PLS_INTEGER := 0;  -- used for counting the actual number of rows returned
...
BEGIN
    connection_id := EXEC_SQL.OPEN_CONNECTION(connect_str);
    cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
    sqlstr := 'select ename, empno, hiredate from emp ';
    EXEC_SQL.PARSE(connection_id, cursorID, sqlstr, exec_sql.V7);
    EXEC_SQL.BIND_VARIABLE(connection_id, cursorID, ':bn', input_empno);
    EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 1, loc_ename, 30);
    EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 2, loc_eno);
    EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 3, loc_hiredate);
    nIgn := EXEC_SQL.EXECUTE(connection_id, cursorID);
    -- call FETCH_ROWS to obtain a row. When a row is returned, obtain the values,
    -- and increment the count.
    --
    WHILE (EXEC_SQL.FETCH_ROWS(connection_id, cursorID) > 0 ) LOOP
        nRows := nRows + 1;
        EXEC_SQL.COLUMN_VALUE(connection_id, cursorID, 1, loc_ename;
        EXEC_SQL.COLUMN_VALUE(connection_id, cursorID, 2, loc_eno);
        EXEC_SQL.COLUMN_VALUE(connection_id, cursorID, 3, loc_hiredate);
...
END LOOP;
--
-- The loop terminates when FETCH_ROWS returns 0. This could have happen because
-- the query was incorrect or because there were no more rows. To distinguish
-- between these cases, we keep track of the number of rows returned.
--
-- IF (nRows <= 0) THEN
```

60     Forms Developer and Reports Developer Built-in Package Reference
EXEC_SQL.More_Result_Sets

**Description**  This function applies to non-Oracle connections only. It determines if there is another result set to retrieve for a specified cursor.

**Syntax**

```
FUNCTION EXEC_SQL.More_Result_Sets
((Connid IN CONNTYPE),
 Curs_Id IN CURSTYPE)
RETURN BOOLEAN;
```

**Parameters**

*Connid*  Is the handle to the connection you want to use.

If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

*Curs_Id*  Is the cursor handle to the SQL statement from which you want to fetch.

**Returns**  TRUE or FALSE

**Usage Notes**  If used against Oracle databases, the function always returns FALSE. If a non-Oracle stored procedure has another result set to retrieve, the function initializes the result set and returns TRUE. Use EXEC_SQL.Describe_Column to obtain information about the new result set and EXEC_SQL.Fetch_Rows to retrieve the data, if required.

**EXEC_SQL.More_Result_Sets example**

```
PROCEDURE esmoreresultsets(sqlstr VARCHAR2) IS
  conidODBC EXEC_SQL.CONNTYPE;
  nRes PLS_INTEGER;
  nRows PLS_INTEGER := 0 ;
  curID EXEC_SQL.CURSTYPE;
BEGIN
  -- an ODBC connection string; usually has the form
  '-- username/password@ODBD:dbname'
  conidODBC := EXEC_SQL.OPEN_CONNECTION('connection_str_ODBC');
  curID := EXEC_SQL.OPEN_CURSOR(conidODBC);
```

...
EXEC_SQL.PARSE(conIDODBC, curID, sqlstr, exec_sql.v7);

nRes := EXEC_SQL.EXECUTE(conIDODBC, curID);

-- obtain results from first query in sqlstr
WHILE (EXEC_SQL.FETCH_ROWS(conIDODBC, curID) > 0) LOOP
    nRows := nRows + 1;
    ...
END LOOP;

-- for some non-Oracle databases, sqlstr may contain a batch of
-- queries;
-- MORE_RESULT_SETS checks for additional result sets
-- IF (EXEC_SQL.MORE_RESULT_SETS(conIDODBC, curID)) THEN
--    TEXT_IO.PUT_LINE(' more result sets ');
-- ELSE
--    TEXT_IO.PUT_LINE(' no more result sets ');
-- END IF;

...
EXEC_SQL.CLOSE_CONNECTION(conIDODBC);
END;

EXEC_SQL.Column_Value

Description  This procedure returns the value of the cursor for a given position in a given
cursor. It is used to access the data fetched by calling EXEC_SQL.Fetch_Rows.

Syntax
PROCEDURE EXEC_SQL.Column_Value
  [(Connid         IN CONNTYPE),
   Curs_Id        IN CURSTYPE,
   Position       IN PLS_INTEGER,
   Value          OUT <datatype>,
   [Column_Error   OUT NUMBER],
   [Actual_Length  OUT PLS_INTEGER]);

where <datatype> is one of the following:
NUMBER
DATE
VARCHAR2

Parameters
Name          Mode     Description
Connid    IN       Is the handle to the connection you want to use. If you do not
                 specify a connection, EXEC_SQL.Default_Connection
                 retrieves the primary Forms Developer connection handle
                 from the cache.
Curs_Id    IN       Is the cursor handle to the row from which you want to get the
**Position**  
IN  
Is the relative position of the column in the specified cursor.  
Starting from the left, the first column is position 1.

**Value**  
OUT  
Returns the value of the specified column and row.

**Column_Error**  
OUT  
Returns the error code for the specified column value (Oracle data sources only).

**Actual_Length**  
OUT  
Returns the actual length of the column value before truncation.

**Usage Notes**  
If you specify a value which has a PL/SQL type that is different from what was specified by EXEC_SQL.Define_Column, the exception EXEC_SQL.Value_Error is raised.

**EXEC_SQL.Column_Value example**

```sql
PROCEDURE getData IS  
    connection_id EXEC_SQL.CONNTYPE;  
    cursorID EXEC_SQL.CURSTYPE;  
    sqlstr VARCHAR2(1000);  
    loc_ename VARCHAR2(30);  
    loc_eno NUMBER;  
    loc_hiredate DATE;  
    nIgn PLS_INTEGER;  
    nRows PLS_INTEGER := 0;  
  
BEGIN  
    connection_id := EXEC_SQL.OPEN_CONNECTION(connect_str);  
    cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);  
    sqlstr := 'select ename, empno, hiredate from emp';  
    EXEC_SQL.PARSE(connection_id, cursorID, sqlstr, exec_sql.V7);  
    EXEC_SQL.BIND_VARIABLE(connection_id, cursorID, ':bn', input_empno);  
    EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 1, loc_ename, 30);  
    EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 2, loc_eno);  
    EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 3, loc_hiredate);  
    nIgn := EXEC_SQL.EXECUTE(connection_id, cursorID);  
    -- You must have used DEFINE_COLUMN to define the column data characteristics before  
    -- using COLUMN_VALUE to retrieve the value. Assign the row’s first value to the
```
EXEC_SQL.Variable_Value

Description  This procedure retrieves the output value of a named bind variable at a specified cursor. It also returns the values of bind variables in anonymous PL/SQL blocks.

Syntax

PROCEDURE EXEC_SQL.Variable_Value
    (Connid IN CONNTYPE,
     Curs_Id IN CURSTYPE,
     Name IN VARCHAR2,
     Value OUT <datatype>);

where <datatype> is one of the following:

NUMBER
DATE
VARCHAR2

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connid</td>
<td>IN</td>
<td>Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.</td>
</tr>
<tr>
<td>Curs_Id</td>
<td>IN</td>
<td>Is the cursor handle you want to retrieve the bind variable from.</td>
</tr>
<tr>
<td>Name</td>
<td>IN</td>
<td>Is the name of the bind variable.</td>
</tr>
<tr>
<td>Value</td>
<td>OUT</td>
<td>Returns the value of the bind variable for the specified cursor.</td>
</tr>
</tbody>
</table>

Usage Notes  If you try to retrieve a data type other than what was specified for the bind variable by EXEC_SQL.Bind_Variable, the exception EXEC_SQL.Value_Error is raised.

EXEC_SQL.Variable_Value example

It is assumed that the following procedure, tstbindnum, exists on the server which is specified by the connection string used in OPEN_CONNECTION.

Create or replace procedure tstbindnum (input IN NUMBER, output OUT NUMBER) as
BEGIN
    output := input * 2;
END;

All this procedure does is to take an input number, double its value, and return it in the out variable.

PROCEDURE esvarvalnum (input IN NUMBER) IS
    connection_id EXEC_SQL.CONNTYPE;
    bIsConnected BOOLEAN;
    cursorID EXEC_SQL.CURSTYPE;

64  Forms Developer and Reports Developer Built-in Package Reference
sqlstr VARCHAR2(1000);
nRes PLS_INTEGER;
mynum NUMBER;
BEGIN
  connection_id := EXEC_SQL.OPEN_CONNECTION('connection_string');
cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
sqlstr := 'begin   tstbindnum(:bn1, :bnret);   end;';   -- an
anonymous block
EXEC_SQL.PARSE(connection_id, cursorID, sqlstr, exec_sql.V7);
  -- define input value
  EXEC_SQL.BIND_VARIABLE(connection_id, cursorID, ':bn1', input);
  -- set up output value
  EXEC_SQL.BIND_VARIABLE(connection_id, cursorID, ':bnret', mynum);
nRes := EXEC_SQL.EXECUTE(connection_id, cursorID);
  -- after the statement is executed, we call VARIABLE_VALUE to obtain
  -- the value of
  -- the bind variable :bnret
  EXEC_SQL.VARIABLE_VALUE(connection_id, cursorID, ':bnret', mynum);
EXEC_SQL.CLOSE_CURSOR(connection_id, cursorID);
EXEC_SQL.CLOSE_CONNECTION(connection_id);
END;

EXEC_SQL.Is_Open

Description  Returns TRUE if a specified cursor is currently open on a specified
connection.

Syntax
FUNCTION EXEC_SQL.Is_Open
   ([Connid      IN CONNTYPE],
    Curs_Id     IN CURSTYPE)
RETURN BOOLEAN;

Parameters
Connid  Is the handle to the connection you want to use.
If you do not specify a connection,
EXEC_SQL.Default_Connection retrieves the
primary Forms Developer connection handle
from the cache.

Curs_Id  Is the cursor handle you want to determine if it
is open.

Returns  TRUE or FALSE
EXEC_SQL.Is_Open, EXEC_SQL.Close_Cursor,
EXEC_SQL.Is_Connected and EXEC_SQL.Close_Connection
example

/*
** This example illustrates the use of EXEC_SQL.Is_Open,
** EXEC_SQL.Close_Cursor, EXEC_SQL.Is_Connected and
** EXEC_SQL.Close_Connection.
*/
PROCEDURE esclosecursor.pld IS
    connection_id EXEC_SQL.CONNTYPE;
    bIsConnected BOOLEAN;
    cr1 EXEC_SQL.CURSTYPE;
    sqlstr1 VARCHAR2(200);
    sqlstr2 VARCHAR2(200);
    nRes PLS_INTEGER;
    bOpen BOOLEAN;
    nRows PLS_INTEGER;
    loc_ename VARCHAR2(30);
    loc_eno NUMBER;
    loc_hiredate DATE;
BEGIN
    BEGIN
        connection_id := EXEC_SQL.OPEN_CONNECTION('connection_str');
        EXCEPTION
            WHEN EXEC_SQL.PACKAGE_ERROR THEN
                TEXT_IO.PUT_LINE('connection open failed ');
    END;
    --
    -- confirm that connection is valid
    --
    bIsConnected := EXEC_SQL.IS_CONNECTED(connection_id);
    IF bIsConnected = FALSE THEN
        TEXT_IO.PUT_LINE('No present connection to any data source. Please
        connect before retrying.');
        RETURN;
    END IF;
    --
    -- open a cursor and do an update
    --
    cr1 := EXEC_SQL.OPEN_CURSOR(connection_id);
    sqlstr1 := 'update emp set empno = 3600 where empno = 7839';
    EXEC_SQL.PARSE(connection_id, cr1, sqlstr1, exec_sql.V7);
    nRes := EXEC_SQL.EXECUTE(connection_id, cr1);
    --
    -- reuse the same cursor, if open, to do another query.
    --
    sqlstr2 := 'select ename, empno, hiredate from emp ';
    --
    -- use IS_OPEN to check the state of the cursor
    --
    IS (EXEC_SQL.IS_OPEN(connection_id, cr1) != TRUE) THEN
        TEXT_IO.PUT_LINE('Cursor no longer available ');
        RETURN;
    END IF;
    --
-- associate the cursor with another statement, and proceed to do the query.
--
EXEC_SQL.PARSE(connection_id, cr1, sqlstr2, exec_sql.V7);
EXEC_SQL.DEFINE_COLUMN(connection_id, cr1, 1, loc_ename, 30);
EXEC_SQL.DEFINE_COLUMN(connection_id, cr1, 2, loc_eno);
EXEC_SQL.DEFINE_COLUMN(connection_id, cr1, 3, loc_hiredate);
nIgn := EXEC_SQL.EXECUTE(connection_id, cr1);
WHILE (EXEC_SQL.FETCH_ROWS(connection_id, cursorID) > 0 ) LOOP
  nRows := nRows + 1;
  EXEC_SQL.COLUMN_VALUE(connection_id, cr1, 1, loc_ename);
  EXEC_SQL.COLUMN_VALUE(connection_id, cr1, 2, loc_eno);
  EXEC_SQL.COLUMN_VALUE(connection_id, cr1, 3, loc_hiredate);
...
END LOOP;
--
-- close the cursor and connection to free up resources
--
EXEC_SQL.CLOSE_CURSOR(connection_id, cr1);
EXEC_SQL.CLOSE_CONNECTION(connection_id);
END;

EXEC_SQL.Close_Cursor

Description  Closes a specified cursor and releases the memory allocated to it.

Syntax
PROCEDURE EXEC_SQL.Close_Cursor
  ((Connid        IN CONNTYPE),
   Curs_Id       IN OUT CURSTYPE);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connid</td>
<td>IN</td>
<td>Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.</td>
</tr>
<tr>
<td>Curs_Id</td>
<td>IN</td>
<td>Is the cursor handle you want to close.</td>
</tr>
</tbody>
</table>

OUT | Sets to NULL. |

Usage Notes  When you no longer need a cursor, you must close it. Otherwise, you may not be able to open new cursors.
EXEC_SQL.Is_Connected

Description  Returns TRUE if a specified connection handle is currently connected to a data source.

Syntax
FUNCTION EXEC_SQL.Is_Connected
   [Connid    IN CONNTYPE]
RETURN BOOLEAN;

Parameters
Connid  Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

Returns  TRUE or FALSE

EXEC_SQL.Is_OCA_Connection

Description  Returns TRUE if a specified connection handle is for an OCA connection.

Syntax
FUNCTION EXEC_SQL.Is_OCA_Connection
   [Connid    IN CONNTYPE]
RETURN BOOLEAN;

Parameters
Connid  Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

Returns  TRUE or FALSE

EXEC_SQL.Is_OCA_Connection example

PROCEDURE esmoreresultsets(sqlstr VARCHAR2) IS
   conidODBC EXEC_SQL.CONNTYPE;
   nRes PLS_INTEGER;
   nRows PLS_INTEGER := 0 ;
   curID EXEC_SQL.CURSTYPE;
BEGIN
   -- an ODBC connection string
   conidODBC := EXEC_SQL.OPEN_CONNECTION('connection_str_ODBC');
curID := EXEC_SQL.OPEN_CURSOR(conidODBC);
EXEC_SQL.PARSE(conidODBC, curID, sqlstr, exec_sql.v7);
nRes := EXEC_SQL.EXECUTE(conidODBC, curID);
--
-- obtain results from first query in sqlstr
--
WHILE (EXEC_SQL.FETCH_ROWS(conidODBC, curID) > 0) LOOP
  nRows := nRows + 1;
...
END LOOP;
--
-- check whether this is an OCA connection. Does not continue for an Oracle
-- connection.
--
IF (EXEC_SQL.IS_OCA_CONNECTION != TRUE) THEN
  TEXT_IO.PUT_LINE('Not an OCA connection ');
  RETURN;
END IF;
--
-- check for more result sets
--
IF (EXEC_SQL.MORE_RESULT_SETS(conidODBC, curID)) THEN
  TEXT_IO.PUT_LINE(' more result sets ');
ELSE
  TEXT_IO.PUT_LINE(' no more result sets ');
END IF;
...
EXEC_SQL.CLOSE_CONNECTION(conidODBC);
END;

EXEC_SQL.Close_Connection

Description  This procedure releases any resources used by the connection handle and invalidates it.

Syntax
PROCEDURE EXEC_SQL.Close_Connection
  ([Connid IN OUT CONNTYPE]);

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connid</td>
<td>IN</td>
<td>Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.</td>
</tr>
</tbody>
</table>
Usage Notes  If the connection is opened by EXEC_SQL.Open_Connection, EXEC_SQL.Close_Connection also closes the database connection. If it is opened by EXEC_SQL.Default_Connection, EXEC_SQL.Close_Connection does not close the database connection.

It is important to close the connection when you do not need it. If you do not close the connection, the database connection remains open and any memory allocated to the connection, including opened cursors, remain in use. This may result in connection deadlocks.

EXEC_SQL.Last_Error_Position

Description  Returns the byte offset in the SQL statement where an error occurred. The first character in the statement is at position 0.

Syntax
FUNCTION EXEC_SQL.Last_Error_Position
   [Connid   IN CONNTYPE]
RETURN PLS_INTEGER;

Parameters
Connid  Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

Returns  An integer.

Usage Notes  Use this function after EXEC_SQL.PARSE, and before another EXEC_SQL procedure or function. The byte offset at which an error occurred cannot be determined for OCA data sources.

EXEC_SQL.Last_Error_Position example

PROCEDURE eslasterrorpos(sqlstr VARCHAR2) is
   connection_id EXEC_SQL.CONNTYPE;
   cursorID EXEC_SQL.CURSTYPE;
   nErrPos PLS_INTEGER := 0;
   errmsg VARCHAR2(256);
BEGIN
   connection_id := EXEC_SQL.OPEN_CONNECTION('');
   cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
   BEGIN
      -- parsing statement from caller
      --
      EXEC_SQL.parse(connection_id, cursorID, sqlstr, exec_sql.V7);
-- check for error in statement; find out position where statement syntax is in error

EXCEPTION
    WHEN EXEC_SQL.PACKAGE_ERROR THEN
        nErrPos := EXEC_SQL.LAST_ERROR_POSITION(connection_id);
        TEXT_IO.PUT_LINE(' position in text where error occurred ' ||
                        nErrPos);
        errmsg := EXEC_SQL.LAST_ERROR_MESG(connection_id);
        TEXT_IO.PUT_LINE(' error message ' || errmsg);
        RETURN;
END;
--
-- here to execute statement
--

...  

nRes := EXEC_SQL.EXECUTE(connection_id, cursorID);
...

EXEC_SQL.CLOSE_CURSOR(connection_id, cursorID);
EXEC_SQL.CLOSE_CONNECTION(connection_id);
END;

EXEC_SQL.Last_Row_Count

Description  Returns the cumulative number of rows fetched.

Syntax

FUNCTION EXEC_SQL.Last_Row_Count
    [Connid IN CONNTYPE]
RETURN PLS_INTEGER;

Parameters

Connid  Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

Returns  An integer.

Usage Notes  Use this function after calling EXEC_SQL.Fetch_Rows or EXEC_SQL.Execute_And_Fetch. The function returns a zero when used after an EXEC_SQL.Execute call.
EXEC_SQL.Last_Row_Count example

PROCEDURE eslastrowcount is
    connection_id EXEC_SQL.CONNTYPE;
    cursorID EXEC_SQL.CURSTYPE;
    sqlstr VARCHAR2(1000);
    nIgn PLS_INTEGER;
    nRows PLS_INTEGER := 0 ;
    mynum NUMBER;
BEGIN
    connection_id := EXEC_SQL.OPEN_CONNECTION('connection_str');
    cursorID := EXEC_SQL.OPEN_CURSOR(connection_id);
    -- in this query, we order the results explicitly
    --
    sqlstr := 'select empno from emp order by empno';
    EXEC_SQL.PARSE(connection_id, cursorID, sqlstr, exec_sql.V7);
    EXEC_SQL.DEFINE_COLUMN(connection_id, cursorID, 1, mynum);
    nIgn := EXEC_SQL.EXECUTE(connection_id, cursorID);
    LOOP
        nIgn := EXEC_SQL.FETCH_ROWS(connection_id, cursorID);
        --
        -- do whatever processing is desired
        --
        IF (nIgn > 0) THEN
            EXEC_SQL.COLUMN_VALUE(connection_id, cursorID, 1, mynum);
            ...
        END IF;
    END LOOP;
    nRows := EXEC_SQL.LAST_ROW_COUNT(connection_id);
    -- In this example, we are only interested in the first 10 rows, and
    exit after
    -- fetching them
    --
    IF (nRows > 10) THEN
        EXIT;
    END IF;
END LOOP;
EXEC_SQL.CLOSE_CURSOR(connection_id, cursorID);
EXEC_SQL.CLOSE_CONNECTION(connection_id);
END;

EXEC_SQL.Last_SQL_Function_Code

Description  Returns the last SQL function code, indicating the type of SQL statement.
For a list of valid function codes, see your Programmer’s Guide to the Oracle Call Interface.

Syntax
FUNCTION EXEC_SQL.Last_SQL_Function_Code
    [Connid IN CONNTYPE]
RETURN PLS_INTEGER;

Parameters
Connid Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

Returns An integer.
Usage Notes Use this function immediately after parsing the SQL statement.

EXEC_SQL.Last_SQL_Function_Code example
/*
 ** In this procedure, a statement is passed in and executed. If the statement is a select statement, then further processing is initiated to determine the column characteristics.
 */
PROCEDURE eslastfunccode(sqlstr VARCHAR2) IS
 connection_id EXEC_SQL.CONNTYPE;
 cursor_number EXEC_SQL.CursType;
  --
  -- The values for the function codes is dependent on the RDBMS version.
  --
  SELECTFUNCCODE PLS_INTEGER := 3;
  sql_str VARCHAR2(256);
  nColumns PLS_INTEGER := 0;
  nFunc PLS_INTEGER := 0;
  colName VARCHAR2(30);
  colLen PLS_INTEGER;
  colType PLS_INTEGER;
BEGIN
  connection_id := EXEC_SQL.OPEN_CONNECTION('connection_str');
  cursor_number := EXEC_SQL.OPEN_CURSOR(connection_id);
  EXEC_SQL.PARSE(connection_id, cursor_number, sql_str, exec_sql.V7);
  nIgnore := EXEC_SQL.EXECUTE(connection_id, cursor_number);
  --
  -- check what kind of function it is
  --
  nFunc := EXEC_SQL.LAST_SQL_FUNCTION_CODE(connection_id);
  IF (nFunc != SELECTFUNCCODE) THEN
    RETURN;
  END IF;
  --
  -- proceed to obtain the column characteristics
  --
  LOOP
    nColumns := nColumns + 1;
    BEGIN
      EXEC_SQL.DESCRIBE_COLUMN(connection_id, cursor_number,
nColumns, colName, colLen, colType);
    TEXT_IO.PUT_LINE(' col= ' || nColumns || ' name ' || colName
    || ' len= ' || colLen || ' type ' || colType
    );
    EXCEPTION
    WHEN EXEC_SQL.INVALID_COLUMN_NUMBER THEN
      EXIT;
    END;
END LOOP;
EXEC_SQL.CLOSE_CURSOR(connection_id, cursor_number);
EXEC_SQL.LCOSE_CONNECTION(connection_id);
END;

EXEC_SQL.Last_Error_Code

Description  Returns the last Oracle error code raised on a connection.

Syntax
FUNCTION EXEC_SQL.Last_Error_Code
[ Connid      IN CONNTYPE]
RETURN PLS_INTEGER;

Parameters
Connid  Is the handle to the connection you want to use. If you do
       not specify a connection, EXEC_SQL.Default_Connection
       retrieves the primary Forms Developer connection handle
       from the cache.

Returns  An integer.

Usage Notes  Use this function immediately after the EXEC_SQL.Package_Error
              exception is raised.

EXEC_SQL.Last_Error_Code and EXEC_SQL.Last_Error_Mesg

example

/*
 ** In the following procedure, we execute a statement that is passed in.
 ** If there
 ** are any exceptions shown, we check to see its nature using
 ** LAST_ERROR_CODE
 ** and LAST_ERROR_MESG.
*/

procedure eslastfunccode(sqlstr varchar2) is
  connection_id exec_sql.connType;
  cursor_number exec_sql.CursType;
  sql_str VARCHAR2(256);

Forms Developer and Reports Developer Built-in Package Reference
nIgnore pls_integer;
BEGIN
  connection_id := exec_sql.open_connection('connection_str');
  cursor_number := exec_sql.open_cursor(connection_id);
  exec_sql.parse(connection_id, cursor_number, sql_str, exec_sql.V7);
  nIgnore := exec_sql.execute(connection_id, cursor_number);
  exec_sql.close_cursor(connection_id, cursor_number);
  exec_sql.close_connection(connection_id);
--
-- check the error in the exception block
--
EXCEPTION
  WHEN exec_sql.package_error THEN
    text_io.put_line('error : ' || 
      to_char(exec_sql.last_error_code(connection_id)) || ' ' || 
      exec_sql.last_error_mesg(connection_id));
--
-- ensure that even though an error has occurred, the cursor and 
-- connection
-- are closed.
--
  IF exec_sql.is_connected(connection_id) THEN
    IF exec_sql.is_open(connection_id, cursor_number) THEN
      exec_sql.close_cursor(connection_id, cursor_number);
    END IF;
    exec_sql.close_connection(connection_id);
  END IF;
END;
EXEC_SQL.Last_Error_Mesg

Description Returns the text message of the last error code raised on a connection.

Syntax

```sql
FUNCTION EXEC_SQL.Last_Error_Mesg
    [Connid IN CONNTYPE]
RETURN VARCHAR2;
```

Parameters

- **Connid**
  
  Is the handle to the connection you want to use. If you do not specify a connection, EXEC_SQL.Default_Connection retrieves the primary Forms Developer connection handle from the cache.

Returns A string.

Usage Notes Use this function immediately after the EXEC_SQL.Package_Error exception is raised.

Tip

To obtain the number of columns in a result set, loop through the columns from 1 until the EXEC_SQL.Invalid_Column_Number exception is raised.

Changing the primary database connection

If you change the primary Forms Developer connection after you have called EXEC_SQL.Default_Connection, the next EXEC_SQL.Default_Connection call continues to return the handle from the cache; it does not automatically return a handle to the new primary connection.

To make sure you have the correct handle, always use EXEC_SQL.Close_Connection (without arguments) before you change the primary connection. This allows EXEC_SQL to free up the memory resources allocated to the previous connection, without actually closing it.
List Package

List package
List.Appenditem
List.Destroy
List.Deleteitem
List.Fail
List.GetItem
List.Insertitem
List.Listofchar
List.Make
List.Nitems
List.Prependitem

List.Appenditem

Description  Appends an item to the List.
Syntax
PROCEDURE List.Appenditem
  (List Listofchar,
   item VARCHAR2);
Parameters
  List  A List.
  item  A List Item.

List.Appenditem example

  /*
  ** Add an item to the end of ‘my_List’ then
  ** print out the number of items in the List
  */
  PROCEDURE append (my_List List.Listofchars) IS
BEGIN
  List.Appenditem(my_List, 'This is the last item.');
  Text_IO.Put_Line(List.GetItem(my_List,
    List.Nitems(my_List)));
END;

List.Destroy

Description  Destroys an entire List.
Syntax
PROCEDURE List.Destroy
  (List Listofchar);
Parameters
  List  A List.

List.Destroy example
  /*
  ** Destroy the List
  */
  List.Destroy(my_package.my_List);

List.Deleteitem

Description  Deletes the item at the specified position in the List.
Syntax
PROCEDURE List.Deleteitem
  (List Listofchar),
  pos  PLS_INTEGER);
Parameters
  List  A List.
  pos  A position (base equals 0).

List.Deleteitem example
  /*
  ** Delete the third item from 'my_List'
  */
  List.Deleteitem(my_package.my_List, 2);
List.Fail

Description  Raised when any List Operation fails.
Syntax
     List.Fail  EXCEPTION;

List.Fail example

    /*
     ** Provide an exception handler for the
     ** List.Fail exception
     */
     EXCEPTION
       WHEN List.Fail THEN
          Text_IO.Put_Line('list Operation Failed');

List.GetItem

Description  Gets an item from the List.
Syntax
     FUNCTION List.GetItem
       (List Listofchar,
        pos  PLS_INTEGER)
     RETURN VARCHAR2;
Parameters
       List        A List.
       pos        A position (base equals 0).

Returns  An item from the specified List.

List.GetItem example

    /*
     ** Retrieve and print out the second item
     ** in my_List
     */
     Text_IO.Put_Line(List.GetItem(my_List, 1));

List.InsertItem

Description  Inserts an item into the List At the specified position.
Syntax
PROCEDURE List.Insertitem
    (List Listofchar),
    pos PLS_INTEGER,
    item VARCHAR2);

Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>A List.</td>
</tr>
<tr>
<td>pos</td>
<td>A position (base equals 0).</td>
</tr>
<tr>
<td>item</td>
<td>A List Item.</td>
</tr>
</tbody>
</table>

List.Insertitem example

/*
** Add a string to the List In third place
** then retrieve the third item and print it
*/
PROCEDURE insert_item(my_List List.Listofchar) IS
    BEGIN
        List.Insertitem(my_List, 2, 'This is the third item.');
        Text_IO.Put_Line(List.GetItem(my_List, 2));
    END;

List.Listofchar

Description  Specifies a handle to a List.

Syntax

TYPE List.Listofchar;

List.Listofchar example

/*@
** Declare a variable of the type
** Listofchar, then create the List
*/
PROCEDURE my_proc IS
    BEGIN
        my_List := List.Make;
    END;

List.Make

Description  Creates a new, empty List.  A List Must be created before it can be used.
Syntax
FUNCTION List.Make
RETURN Listofchar;

Usage Notes Any Lists created with this function should be destroyed with the List.Destroy procedure.

List.Make example

```
/*
  ** Create a List Of the type Listofchar
*/
PROCEDURE my_proc IS
  my_List List.Listofchar;
BEGIN
  my_List := List.Make;
END;
```

List.Nitems

Description Returns the number of items in the List.

Syntax
FUNCTION List.Nitems
  (List Listofchar)
RETURN PLS_INTEGER;

Parameters

List       A List.

List.Nitems example

```
/*
  ** For each item in my_List, retrieve the
  ** value of the item and print it out
*/
PROCEDURE print_List Is
BEGIN
  FOR i IN 0..List.Nitems(my_pkg.my_List)-1 LOOP
    Text_IO.Put_Line(List.Getitem(my_pkg.my_List, i));
  END LOOP;
END;
```

List.Prependitem

Description Adds a List Item to the beginning of a List.
**Syntax**

PROCEDURE List.PrependItem  
(List Listofchar),  
(item VARCHAR2);

**Parameters**

<table>
<thead>
<tr>
<th>List</th>
<th>A List.</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>A List Item.</td>
</tr>
</tbody>
</table>

**List.PrependItem example**

```
/*
** Insert a string to the beginning of my_List
** then retrieve it and print it out
*/
PROCEDURE prepend(my_List List.Listofchars) IS  
BEGIN  
    List.PrependItem(my_List, 'This is the first item.');  
    Text_IO.Put_Line(List.GetItem(my_List, 0));  
END;
```

82     Forms Developer and Reports Developer Built-in Package Reference
OLE2 Package

OLE2 package

OLE2.Add_Arg
OLE2.Add_Arg_Obj
OLE2.Create_Arglist
OLE2.Create_Obj
OLE2.Destroy_Arglist
OLE2.Get_Char_Property
OLE2.Get_Num_Property
OLE2.Get_Obj_Property
OLE2.Invoke
OLE2.Invoke_Num
OLE2.Invoke_Char
OLE2.Invoke_Obj
OLE2.IsSupported
OLE2.Last_Exception
OLE2.List_Type
OLE2.Obj_Type
OLE2.OLE_Error
OLE2.OLE_Not_Supported
OLE2.Release_Obj
OLE2.Set_Property

OLE2.Add_Arg

Description  Appends an argument to an argument List created with OLE2.Create_Arglist.

Syntax

PROCEDURE OLE2.Add_Arg
  (List   List_Type,
   value  NUMBER);
PROCEDURE OLE2.Add_Arg
  (List   List_Type,
   value  VARCHAR2);

Parameters
  List        The name of an argument List assigned to the
              OLE2.Create_Arglist Function.
  value       The argument value.

Usage Notes  The argument can be of the type NUMBER or VARCHAR2.

OLE2.Add_Arg example

  /*
   ** Add an argument to my_Arglist
   */
  OLE2.Add_Arg(my_Arglist, 'Sales Revenue');

---

OLE2.Add_Arg_Obj

Description  Appends an object argument to an argument list created with
              OLE2.Create_Arglist.

Syntax
  PROCEDURE OLE2.Add_Arg_Obj
    (List IN List_Type,
     value IN Obj_Type);

Parameters
  List        A list handle returned from a call to the
              OLE2.Create_Arglist function.
  value       The value of an Obj_Type argument to be
              passed to the OLE2 automation server.
OLE2.Add_Arg_Obj example

/*
** When the OLE interface must accept an unknown object
** as an argument instead of a pure scalar type, use the
** Add_Arg_Obj procedure.
*/
object = OLE2.CREATE_OBJ(obj_name);
listh := OLE2.CREATE_ARGLIST;
OLE2.ADD_ARG_OBJ(listh, object);

OLE2.Create_Arglist

Description  Creates an argument List you can pass to the OLE2 Automation server.

Syntax
FUNCTION OLE2.Create_Arglist
RETURN List_Type;

Returns  A handle to an argument List.

OLE2.Create_Arglist example

/*
** Declare a variable of the type OLE2.List_Type
** then create the argument List Of that name
*/
my_Arglist  OLE2.List_Type;
BEGIN
my_Arglist := OLE2.Create_Arglist;
OLE2.Add_Arg(my_Arglist, 'Sales Revenue');
END;

OLE2.Create_Obj

Description  Creates an OLE2 Automation Object.

Syntax
FUNCTION OLE2.Create_Obj
   (object   VARCHAR2)
RETURN obj_type;

Parameters
   object  An OLE2 Automation Object.
Returns  A handle to an OLE2 Automation Object.

**OLE2.Create_Obj example**

```plaintext
/*
** Create an OLE2 Object
*/
obj := OLE2.Create_Obj('Excel.Application.5');
```

**OLE2.Destroy_Arglist**

Description  Destroys an argument List previously created with the OLE2.Create_Arglist Function.

Syntax  
```plaintext
PROCEDURE OLE2.Destroy_Arglist
  (List List_Type);
```

Parameters  
```plaintext
  List
      The name of an argument List assigned to the
      OLE2.Create_Arglist Function.
```

**OLE2.Destroy_Arglist example**

```plaintext
/*
** Destroy an argument list.
*/
OLE2.Destroy_Arglist(My_Arglist);
```

**OLE2.Get_Char_Property**

Description  Gets a property of an OLE2 Automation Object.

Syntax  
```plaintext
FUNCTION OLE2.Get_Char_Property
  (object obj_type,
   property VARCHAR2,
   Arglist List_Type := 0)
RETURN VARCHAR2;
```

Parameters  
```plaintext
  object
      An OLE2 Automation Object.
  property
      The name of a property in an OLE2 Automation Object.
  Arglist
      The name of an argument List assigned to the
```
OLE2.Create_Arglist Function.

**Returns**  A character value.

### OLE2.Get_Char_Property example

```c
/*
** Get the property for the object.
*/
str := OLE2.Get_Char_Property(obj, 'text');
```

---

### OLE2.Get_Num_Property

**Description**  Gets a number value from an OLE2 Automation Object.

**Syntax**

```c
FUNCTION OLE2.Get_Num_Property
    (object    obj_type,
     property  VARCHAR2,
     Arglist   List_Type := 0)
RETURN NUMBER;
```

**Parameters**

- **object**: An OLE2 Automation Object.
- **property**: The name of a property in an OLE2 Automation Object.
- **Arglist**: The name of an argument List assigned to the OLE2.Create_Arglist Function.

**Returns**  A number value.

### OLE2.Get_Num_Property example

```c
/*
** Get the number value for the center of the map.
*/
x := OLE2.Get_Num_Property(Map,'GetMapCenterX');
y := OLE2.Get_Num_Property(Map,'GetMapCenterY');
```

---

### OLE2.Get_Obj_Property

**Description**  Gets an object type value from an OLE2 Automation Object.

**Syntax**

```c
FUNCTION OLE2.Get_Obj_Property
```

---
(object  obj_type,
    property VARCHAR2,
    Arglist  List_Type := 0)
RETURN OBJ_TYPE;

Parameters
object    An OLE2 Automation Object.
property   The name of an OLE2 Automation Object.
Arglist    The name of an argument List assigned to the
            OLE2.Create_Arglist Function.

Returns  An OLE2 Automation Object property.

OLE2.Get_Obj_Property example

/*
**Get the object type value for the spreadsheet object.
*/
the_obj:=OLE2.Get_Obj_Property(spreadsheet_obj,
    ‘Application’);

OLE2.Invoke

Description  Invokes an OLE2 method.
Syntax
PROCEDURE OLE2.Invoke
    (object obj_type,
      method VARCHAR2,
      List    List_Type := 0);

Parameters
object    An OLE2 Automation Object.
method    A method (procedure) of the OLE2 Object.
List      The name of an argument List assigned to the
            OLE2.Create_Arglist Function.

OLE2.Invoke example

/*
**Invoke ZoomIn.
*/
OLE2.Invoke(Map, ‘ZoomIn’, my_Arglist);
OLE2.Invoke_Num

Description  Gets a number value from an OLE2 Automation Object, using the specified method.

Syntax
FUNCTION OLE2.Invoke_Num
    (object    obj_type,
     method   VARCHAR2,
     Arglist  List_Type := 0)
RETURN NUMBER;

Parameters

object    An OLE2 Automation Object.
method    The name of an OLE2 Automation method (function) that returns a number value.
Arglist   The name of an argument List assigned to the OLE2.Create_Arglist Function.

Returns  A number value.

OLE2.Invoke_Num example

    /*
      **  Get the number value using the specified method.
      */
    the_num:=OLE2.Invoke_Num (spreadsheet_obj, 'npv',
        my_Arglist);

OLE2.Invoke_Char

Description  Gets a character value from an OLE2 Automation Object using the specified method.

Syntax
FUNCTION OLE2.Invoke_Char
    (object    obj_type,
     method   VARCHAR2,
     Arglist  List_Type := 0)
RETURN VARCHAR2;

Parameters

object    An OLE2 Automation Object.
method    The name of an OLE2 Automation method (function) that returns a character value.
Arglist   The name of an argument List assigned to the OLE2.Create_Arglist Function.
Returns  A character value.

**OLE2.Invoke_Char example**

```/*
** Get the character value for spell_obj.
*/
correct:=OLE2.Invoke_Char(spell_obj, 'spell', my_Arglist);
```

**OLE2.Invoke_Obj**

**Description**  Gets an object type value from an OLE2 Automation Object.

**Syntax**

FUNCTION OLE2.Invoke_Obj
  (object    obj_type,
   method   VARCHAR2,
   Arglist  List_Type := 0)
RETURN OBJ_TYPE;

**Parameters**

- `object`  An OLE2 Automation Object.
- `method`  The name of an OLE2 Automation method to invoke.
- `Arglist`  The name of an argument List assigned to the OLE2.Create_Arglist Function.

**Returns**  An OLE2 Automation Object.

**OLE2.Invoke_Obj example**

```/*
** Get the object type value for wp_obj.
*/
para_obj:=OLE2.Invoke_Obj(wp_obj, 'get_para', my_Arglist);
```

**OLE2.IsSupported**

**Description**  Confirms that the OLE2 package is supported on the current platform.

**Syntax**

OLE2.IS_SUPPORTED

**Returns**  TRUE, if OLE2 is supported on the platform; FALSE if it is not.
OLE2.IsSupported example

/*
** Before calling an OLE2 object in platform independent code,
** use this predicate to determine if OLE2 is supported on the
** current platform.
*/
IF (OLE2.ISSUPPORTED) THEN
    . . . PL/SQL code using the OLE2 package
ELSE
    . . . message that OLE2 is not supported
END IF;

OLE2.Last_Exception

Description  Returns the last OLE2 exception signaled by a PL/SQL exception.
Syntax
FUNCTION last_exception return NUMBER;
or
FUNCTION last_exception(message OUT VARCHAR2) return NUMBER;
Parameters
message  A text string (VARCHAR2) containing the text of the OLE2 error message.
If included, this variable is returned to the caller of the function, in addition to the error code value.

Returns  The complete OLE2 error code from the last OLE2 exception.
Usage Notes
- You can use either syntax for this function. The first syntax returns only the error code; the second syntax returns a text description of the error, in addition to the error code.
- This function returns a complete OLE2 (Windows) style error code as a NUMBER. To extract just the error code portion, you must remove the highest bit (Severity) and then translate the remaining number to INTEGER or BINARY_INTEGER format. See OLE2.Last_Exception example for an example of a procedure that extracts the error code as an integer.

OLE2.Last_Exception example

PACKAGE olepack IS
    PROCEDURE init(...);
    PROCEDURE something(...);

PROCEDURE shutdown(...);
FUNCTION get_err(message OUT VARCHAR2) RETURN BINARY_INTEGER;
END olepack;

PACKAGE BODY olepack IS
...
FUNCTION get_err(message OUT VARCHAR2) RETURN BINARY_INTEGER IS
  -- OLE errors are formatted as 32 bit unsigned integers and
  -- returned as Oracle NUMBERS. We want to extract only the
  -- error code, which is contained in the lowest 16 bits.
  -- We must first strip off the top [severity] bit, if it
  -- exists. Then, we must translate the error to an
  -- INTEGER or BINARY INTEGER and extract the error code.
  --
  -- Some helpful constants:
  -- 0x80000000 = 2147483648
  -- 0x100000000 = 4294967296
  -- 0x0000FFFF =      65535
  --
  hibit NUMBER := 2147483648;
four_gig NUMBER := 4294967296;
code_mask NUMBER := 65535;
excep NUMBER;
trunc_bi BINARY_INTEGER;
ole_code BINARY_INTEGER;
BEGIN
  excep := OLE2.LAST_EXCEPTION(message);
  IF (excep >= hibit) AND (excep <= four_gig) THEN
    trunc_bi := excep - hibit;
  END IF;
  ole_code := BITAND(trunc_bi, code_mask);
  RETURN ole_code;
END get_err;
END olepack;

PROCEDURE ole_test IS
  err_code BINARY_INTEGER;
  err_text VARCHAR2(255);
BEGIN
  olepack.init(...);
  olepack.something(...);
  olepack.shutdown(...);
EXCEPTION
  WHEN OLE2.OLE_ERROR THEN
    err_code := olepack.get_err(err_text);
    TEXT_IO.PUT_LINE('OLE Error #' || err_code || ' : ' || err_text);
    olepack.shutdown(...);
END ole_test;
OLE2.List_Type

Description  Specifies a handle to an argument List.
Syntax

\[
\text{List} \quad \text{OLE2.LIST\_TYPE};
\]

OLE2.List_Type example

```
...  
alist \quad \text{OLE2.LIST\_TYPE};
...  
  alist := \text{OLE2.CREATE\_ARGLIST};
  \text{OLE2.ADD\_ARG}(alist, \text{<argument1>});
  \text{OLE2.ADD\_ARG}(alist, \text{<argument2>});
...  
  wkbook := \text{OLE2.INVOKE\_OBJ}(\text{my\_obj}, \text{‘method1’}, \text{alist});
...  
```

OLE2.Obj_Type

Description  Specifies a handle to an OLE2 Automation Object.
Syntax

\[
\text{obj} \quad \text{OLE2.OBJECT\_TYPE};
\]

Usage Notes
- OLE2.Obj_Type is the OLE2 package’s equivalent of an unknown object.
- For more information, see FORMS_OLE.GET\_INTERFACE\_POINTER in the Forms documentation.

OLE2.Obj_Type example

```
/*
 ** Create an OLE2 Object
 */

\[
\text{obj} \quad \text{OLE2.OBJECT\_TYPE};
\]

\[
\text{obj} := \text{OLE2.Create\_Obj}(\text{‘Excel.Application.5’});
\]
...  
```
OLE2.OLE_Error

**Description**  This exception is raised on an error in the OLE2 package.

**Syntax**

```sql
OLE2.OLE_ERROR EXCEPTION;
```

**OLE2.OLE_Error example**

```sql
PROCEDURE ole_test IS
  err_code BINARY_INTEGER;
  err_text VARCHAR2(255);
BEGIN
  olepack.init(...);
  olepack.something(...);
  olepack.shutdown(...);
EXCEPTION
  WHEN OLE2.OLE_ERROR THEN
    err_code := olepack.get_err(err_text);
    TEXT_IO.PUT_LINE('OLE Error #' || err_code || ': ' || err_text);
    olepack.shutdown(...);
END ole_test;
```

---

OLE2.OLE_Not_Supported

**Description**  This exception is raised if a call is made to the OLE2 package but OLE2 is not supported on the current software platform.

**Syntax**

```sql
OLE2.OLE_NOT_SUPPORTED EXCEPTION;
```

**OLE2.OLE_Not_Supported example**

```sql
PROCEDURE ole_test IS
  err_code BINARY_INTEGER;
  err_text VARCHAR2(255);
BEGIN
  olepack.init(...);
  olepack.something(...);
  olepack.shutdown(...);
EXCEPTION
  WHEN OLE2.OLE_NOT_SUPPORTED THEN
    TEXT_IO.PUT_LINE('OLE2 is not supported on this computer');
    olepack.shutdown(...);
END ole_test;
```
OLE2.Release_Obj

**Description**  Signals an OLE2 Automation Object that the PL/SQL client no longer needs it.

**Syntax**

```
PROCEDURE OLE2.Release_Obj
    (object    obj_type);
```

**Parameters**

- `object`  An OLE2 Automation Object.

**Usage Notes**  This allows the operating system to deallocate any resources related to the object. You must release each OLE2 Automation Object you create or invoke using the OLE2 Package.

**OLE2.Release_Obj example**

```
/*
**Release the OLE2 object objap.
*/
objap   OLE2.Obj_Type
objap:=OLE2.Create_Obj('Excel.application.5');
OLE2.Release_Obj(objap);
```

OLE2.Set_Property

**Description**  Sets the value of a property of an OLE2 Automation Object.

**Syntax**

```
PROCEDURE OLE2.Set_Property
    (object    obj_type,
     property  VARCHAR2,
     value     NUMBER,
     Arglist   List_Type := 0);
```

```
PROCEDURE OLE2.Set_Property
    (object    obj_type,
     property  VARCHAR2,
     value     VARCHAR2,
     Arglist   List_Type := 0);
```

**Parameters**

- `object`  An OLE2 Automation Object.
- `property`  The name of a property in an OLE2 Automation Object.
- `value`  A property value.
- `Arglist`  The name of an argument List assigned to the OLE2.Create_Arglist Function.
OLE2.Set_Property example

/*
 **Set properties for the OLE2 object ‘Excel.Application’.
 */
application:=OLE2.CREATE_OBJ('Excel.Application');
OLE2.Set_Property(application,'Visible', 'True');

workbooks:=OLE2.INVOKE_OBJ(application, 'Workbooks');
workbook:=OLE2.INVOKE_OBJ(workbooks,'Add');
worksheets:=OLE2.INVOKE_OBJ(workbook, 'Worksheets');
worksheet:=OLE2.INVOKE_OBJ(worksheets,'Add');
args:=OLE2.CREATE_ARGLIST;
OLE2.ADD_ARG(args, 4);
OLE2.ADD_ARG(args, 2);
cell:=OLE2.Invoke_Obj(worksheet, 'Cells', args);
OLE2.DESTROY_ARGLIST(args);
OLE2.Set_Property(cell, 'Value', 'Hello Excel!');
**Ora_Ffi Package**

**Ora_Ffi package**

- Ora_Ffi.Ffi_Error
- Ora_Ffi.Find_Function
- Ora_Ffi.Find_Library
- Ora_Ffi.Funchandletype
- Ora_Ffi.Generate_Foreign
- Ora_Ffi.Is_Null_Ptr
- Ora_Ffi.Libhandletype
- Ora_Ffi.Load_Library
- Ora_Ffi.Pointertype
- Ora_Ffi.Register_Function
- Ora_Ffi.Register_Parameter
- Ora_Ffi.Register_Return
- Ora_Ffi.Unload_Library
- Ora_Ffi Example 1A
- Ora_Ffi Example 1B
- Ora_Ffi Example 2

**Ora_Ffi.Ffi_Error**

**Description**  Raised when an error occurs while using the Ora_Ffi Package.

**Syntax**

```
EXCEPTION  Ora_Ffi_Error;
```

**Ora_Ffi.Ffi_Error example**

```
/* This example uses Ora_Ffi_Error */
PROCEDURE register_libs IS
  testlib_lhandle ora_ffi.libhandletype;
BEGIN
```

Forms Developer and Reports Developer Built-in Package Reference 97
/* Attempt to load a dll library from a non-existant directory*/
testlib_lhandle := ora_ffl.load_library
('C:\baddir\', 'libtest.dll');

EXCEPTION
WHEN Ora_Ffi.Ffi_Error THEN
  /* print error message */
  text_io.put_line(tool_err.message);
  /* discard the error */
  tool_err.pop;
END;

---

Ora_Ffi.Find_Function

**Description**
Locates and returns the function handle for the specified function. You can retrieve the function handle by specifying either a function name or a library name. The function must previously have been registered with Ora_Ffi.Register_Function.

**Syntax**

FUNCTION Ora_Ffi.Find_Function
  (libHandle     libHandleType,
   funcname      VARCHAR2)
RETURN funcHandleType;

FUNCTION Ora_Ffi.Find_Function
  (libname       VARCHAR2,
   funcname      VARCHAR2)
RETURN funcHandleType;

**Parameters**

- **libHandle**: A library handle returned by Ora_Ffi.Load_Library or Ora_Ffi.Find_Library.
- **funcname**: The name of the function to be located.
- **libname**: The name of the library the function is in.

**Returns**
A handle to the specified function.

**Ora_Ffi.Find_Function example**

/* Find foreign function handle for a given foreign library handle and foreign function name */

BEGIN
  funchandle := ora_ffl.find_function
               (libhandle, 'my_func');
  ...

---

98 Forms Developer and Reports Developer Built-in Package Reference
/* Find foreign function handle for a given foreign function and foreign library names */
BEGIN
  funchandle := ora_ffi.find_function
               (libhandle, 'my_func);
  ...
END;

**Ora_Ffi.Find_Library**

**Description**  Locates and returns the handle for the specified foreign library name. The library must previously have been registered with Ora_Ffi.Load_Library.

**Syntax**

```sql
FUNCTION Ora_Ffi.Find_Library (libname VARCHAR2) RETURN libHandleType;
```

**Parameters**

- `libname`  The name of the library.

**Returns**  A handle to the specified foreign library.

**Ora_Ffi.Find_Library example**

```sql
/* Find foreign library handle for a given library name */
BEGIN
  ...
  libhandle := ora_ffi.find_library
               ('mylib.dll');
  ...
END;
```

**Ora_Ffi.Funchandletype**

**Description**  Specifies a handle to a foreign function. You can use Ora_Ffi.Find_Function to obtain the handle.

**Syntax**

```sql
TYPE Ora_Ffi.Funchandletype;
```
Ora_Ffi.Funchandletype example

/* This example uses Ora_Ffi.Funchandletype */

PROCEDURE define_c_funcs IS
    getresult_fhandle  ora_ffi.funcHandleType;
    foo_fhandle        ora_ffi.funcHandleType;
BEGIN
    /* Register the info for function getresult */
    getresult_fhandle := ora_ffi.register_function
        (testlib_lhandle, 'getresult');
    ...
    /* Register the info for function foo */
    foo_fhandle := ora_ffi.register_function
        (testlib_lhandle,'foo');
    ...
END;

Ora_Ffi.Generate_Foreign

Description Generates a package of PL/SQL code for all the functions defined in the specified library. You must first load the library, register all of the functions you want to invoke, and register their parameter and return values.

Syntax

PROCEDURE Ora_Ffi.Generate_Foreign
    (handle  libHandleType);
PROCEDURE Ora_Ffi.Generate_Foreign
    (handle   libHandleType,
     pkgname  VARCHAR2);

Parameters

handle A library handle returned by Ora_Ffi.LoadLibrary or Ora_Ffi.Find_Library.

pkgname The name of the package to be generated.

If you do not specify a package name, the name of the library, prefixed with FFI_, is used. For example, if the library name is LIBTEST, the package name will be FFI_LIBTEST.

Usage Notes

- Packages generated by the Ora_Ffi.Generate.Foreign function are created in your current name space and will appear under the Program Units node of the Procedure Builder Object Navigator. Once a package has been generated, you can copy it to the Program Units node of a PL/SQL Library or to the Stored Program Units node of a database, and you can export it to a text file using File->Export, just like any other new package or procedure that you have defined.
A PL/SQL package generated by the Ora_Ffi.Generate_Foreign function automatically includes the required PRAGMA compiler directives for each of the registered functions:

PRAGMA interface (C, func_name, 11265);

where func_name is the name of a registered foreign function from a dll library that has already been loaded. You can specify the name of the generated PL/SQL package, but within that package, each of the entry points will match the names of the foreign functions they map to.

**Ora_Ffi.Generate_Foreign example**

```plsql
/* Define components of package test */
PACKAGE test IS
  ...
END;

/* Define package body procedures */
PACKAGE BODY test IS
  PROCEDURE register_libs IS
    BEGIN
      /* Load the test library */
      testlib_lhandle := Ora_Ffi_.load_library
        ('c:\orawin95\oralibs\','testlib.dll')
    END;

  PROCEDURE define_c_funcs IS
    getresult_fhandle Ora_Ffi.Funchandletype;
    foo_handle        Ora_Ffi.Funchandletype;
    BEGIN
      /* Register the info for function getresult */
      getresult_fhandle := ora_ffi.register_function
        (testlib_lhandle,'getresult');
      ...
      /* Register the info for function foo */
      foo_fhandle := ora_ffi.register_function
        (testlib_lhandle,'foo');
      ...
      /* Generate PL/SQL package containing all functions defined in test library */
      ora_ffi.generate_foreign
        (testlib_lhandle, 'test_ffi_pkg');
      ...
    END;
END;
END;
```

**Ora_Ffi.Is_Null_Ptr**

**Description**  Determines whether a library, function, or pointer handle is null.

**Syntax**  

Forms Developer and Reports Developer Built-in Package Reference 101
FUNCTION Ora_Ffi.Is_Null_Ptr (handle libHandleType) RETURN BOOLEAN;
FUNCTION Ora_Ffi.Is_Null_Ptr (handle funcHandleType) RETURN BOOLEAN;
FUNCTION Ora_Ffi.Is_Null_Ptr (handle pointerType) RETURN BOOLEAN;

Parameters
handle The library, function, or pointer to evaluate.

Returns
TRUE If the handle is null.
FALSE If the handle is not null.

Ora_Ffi.Is_Null_Ptr example
/* This example uses Ora_Ffi.Is_Null_Ptr */
PROCEDURE register_libs IS ...
BEGIN
/* Load foreign function library */
libhandle := Ora_Ffi.load_library
('C:\oralibs\', 'libfoo.dll');
/* Test whether library is null */
IF (ora_ffi.is_null_ptr(libhandle)) THEN ...
END;

Ora_Ffi.LibHandleType

Description Specifies a handle to a foreign function. Use Ora_Ffi.Find_Function to obtain the handle.

Syntax
TYPE Ora_Ffi.LibHandleType;

Ora_Ffi.LibHandleType example
/* This example uses Ora_Ffi.LibHandleType */
PACKAGE test is
/* Specify that testlib_lhandle is a library handle variable type */
testlib_lhandle ora_ffi.libHandleType;
...
END;
PACKAGE BODY test IS

102 Forms Developer and Reports Developer Built-in Package Reference
PROCEDURE register_libs IS
BEGIN
    testlib_lhandle := Ora_Ffi.Load_library
       ('C:\libdir\','test.dll');
...
END;
...
END;

Ora_Ffi.Load_Library

Description  Loads a specified dynamic library so that its functions can be registered.

Syntax
FUNCTION Ora_Ffi.Load_Library
       (dirname VARCHAR2,
        libname VARCHAR2)
RETURN libHandleType;

Parameters
   dirname          The directory in which the library is located.
   libname          The filename of the library.

Returns  A handle to the foreign library. It returns a null handle if the library was unable to be found or loaded.

Ora_Ffi.Load_Library example

/* This example uses Ora_Ffi.Load_Library */

PACKAGE test is
   /* Declare testlib_lhandle as an Ora_Ffi
      library handle variable type. */
   testlib_lhandle ora_ffi.libHandleType;
...
END;

PACKAGE BODY test IS
PROCEDURE register_libs IS
BEGIN
   /* Load the dynamic link library ‘test.dll’
      from the directory C:\libdir\ and return
      the handle testlib_lhandle. */
   testlib_lhandle := Ora_Ffi.Load_library
       ('C:\libdir\','test.dll');
...
END;
...
END;
Ora_Ffi.Pointertype

Description  Can assume the value of a generic C pointer (i.e., a pointer of unspecified type).

Syntax
TYPE Ora_Ffi.Pointertype;

Ora_Ffi.Pointertype example

/* This example uses Ora_Ffi.Pointertype */

PACKAGE imglib IS
  /* Declare Function get_image which returns a generic C pointer. */
  FUNCTION get_image(ikey IN OUT VARCHAR2) RETURN Ora_Ffi.Pointertype;
  /* Declare Procedure show_image with parameter idata which is a generic C pointer. */
  PROCEDURE show_image(idata Ora_Ffi.Pointertype,
                        iscale NUMBER);
END;
...

PROCEDURE display_image(keywrd IN OUT VARCHAR2) IS
  /* Declare img_ptr as a generic C pointer type */
  img_ptr Ora_Ffi.Pointertype;
BEGIN
  img_ptr := imglib.get_image(keywrd);
  imglib.show_image(img_ptr,2);
END;

Ora_Ffi.Register_Function

Description  Registers a specified foreign function.

Syntax
FUNCTION Ora_Ffi.Register_Function (libHandle libHandleType,
                   funcname VARCHAR2,
                   callstd NUMBER := C_STD) RETURN funcHandleType;

Parameters

<table>
<thead>
<tr>
<th>libHandle</th>
<th>A library handle returned by Ora_Ffi.Load_Library or Ora_Ffi.Find_Library.</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcname</td>
<td>The name of the function to be registered.</td>
</tr>
<tr>
<td>callstd</td>
<td>The calling used by the foreign function. (For</td>
</tr>
</tbody>
</table>

104 Forms Developer and Reports Developer Built-in Package Reference
more information, refer to your compiler documentation.) The value of this argument may be one of the following packaged constants:

**C_STD**  Means the foreign function uses the C calling standard.

**PASCAL_STD**  Means the foreign function uses the Pascal calling standard.

**Returns**  A handle to the foreign function.

### Ora_Ffi.Register_Function example

```sql
/* Define Procedure define_cfuncs which calls two Ora_Ffi functions, getresult and foo. */
PROCEDURE define_cfuncs is
  getresult_fhandle ora_ffi.funcHandleType;
  foo_fhandle        ora_ffi.funcHandleType;
BEGIN
  /* Register the info for function getresult */
  getresult_fhandle := ora_ffi.register_function
    (testlib_lhandle, 'getresult');
  ...

  /* Register the info for function foo */
  foo_fhandle := ora_ffi.register_function
    (testlib_lhandle, 'foo');
  ...

  /* Generate PL/SQL package containing all functions defined in test library */
  ora_ffi.generate_foreign
    (testlib_lhandle, 'test_ffi_pkg');
END;
```

### Ora_Ffi.Register_Parameter

**Description**  Registers the argument type of the current argument of the specified foreign function.

**Syntax**

```sql
PROCEDURE Ora_Ffi.Register_Parameter
  (funcHandle funcHandleType,
   cargtype   PLS_INTEGER);
```
PROCEDURE Ora_Ffi.Register_Parameter
(funcHandle funcHandleType,
cargtype PLS_INTEGER,
plsrargtype PLS_INTEGER);

Parameters

funcHandle A function handle returned by Ora_Ffi.Register_Function or Ora_Ffi.Find_Function.
cargtype The C datatype of the current argument to the C foreign function being called. The value of this argument may be one of the following packaged constants:

- C_CHAR Means char
- C_CHAR_PTR Means char *
- C_DOUBLE Means double
- C_DOUBLE_PTR Means double *
- C_FLOAT Means float
- C_FLOAT_PTR Means float *
- C_INT Means int
- C_INT_PTR Means int *
- C_LONG Means long
- C_LONG_PTR Means long *
- C_SHORT Means short
- C_SHORT_PTR Means short *
- C_VOID_PTR Means void *

plsrargtype The corresponding PL/SQL argument type (optional).

Ora_Ffi.Register_Parameter example

/* Define Procedure define_c_funcs which calls two Ora_Ffi functions, getresult and foo. */
PROCEDURE define_c_funcs is
getresult_fhandle ora_ffi.funcHandleType;
foo_fhandle ora_ffi.funcHandleType;
BEGIN
/* Register the info for function getresult */
getresult_fhandle := ora_ffi.register_function
(testlib_lhandle,'getresult');
...
/* Register the info for function foo */
foo_fhandle := ora_ffi.register_function
(testlib_lhandle,'foo');
/* Register the return type for function foo */
ora_ffi.register_return
(foo_fhandle, ora_ffi.C_SHORT);
/* Register the parameter info for function foo */
ora_ffi.register_parameter
  (foo_fhandle, ora_ffi.C_FLOAT);
ora_ffi.register_parameter
  (foo_fhandle, ora_ffi.C_INT);
ora_ffi.register_parameter
  (foo_fhandle, ora_ffi.C_CHAR_PTR);

/* Generate PL/SQL package containing all functions
defined in test library */
ora_ffi.generate_foreign
  (testlib_lhandle, 'test_ffi_pkg');
...
END;

Ora_Ffi.Register_Return

Description  Registers the return type of the specified foreign function.

Syntax
PROCEDURE Ora_Ffi.Register_Return
  (funcHandle   funcHandleType,
   creturntype  PLS_INTEGER);
PROCEDURE Ora_Ffi.Register_Return
  (funcHandle   funcHandleType,
   creturntype  PLS_INTEGER,
   plsreturntype PLS_INTEGER);

Parameters

  funcHandle  A function handle returned by
              Ora_Ffi.Register_Function or
              Ora_Ffi.Find_Function.

  creturntype  The C datatype returned by the foreign
               function.  The value of this argument
               may be one of the following packaged constants:
               C_CHAR    Means char
               C_CHAR_PTR Means char *
               C_DOUBLE  Means double
               C_DOUBLE_PTR Means double *
               C_FLOAT   Means float
               C_FLOAT_PTR Means float *
               C_INT     Means int
               C_INT_PTR  Means int *
               C_LONG    Means long
               C_LONG_PTR Means long *
               C_SHORT   Means short
C_SHORT_PTR  Means short *
C_VOID_PTR  Means void *

Ora_Ffi.Register_Return example

/* Define Procedure define_c_funcs which calls two 
Ora_Ffi functions, getresult and foo. */

PROCEDURE define_c_funcs is
    getresult_fhandle  ora_ffi.funcHandleType;
    foo_fhandle        ora_ffi.funcHandleType;

BEGIN
    /* Register the info for function getresult */
    getresult_fhandle := ora_ffi.register_function
        (testlib_lhandle,'getresult');
    /* Register the return type for function getresult */
    ora_ffi.register_return
        (getresult_fhandle, ora_ffi.C_CHAR_PTR);

    /* Register the info for function foo */
    foo_fhandle := ora_ffi.register_function
        (testlib_lhandle,'foo');
    /* Register the return type for function foo */
    ora_ffi.register_return
        (foo_fhandle, ora_ffi.C_SHORT);
    ...
    /* Generate PL/SQL package containing all 
    functions defined in test library */
    ora_ffi.generate_foreign
        (testlib_lhandle, 'test_ffi_pkg');
    ...
END;

Ora_Ffi.Unload_Library

Description  Unloads the specified dynamic library. The functions in the library will no longer be accessible until the library is loaded again.

Syntax
PROCEDURE Ora_Ffi.Unload_Library
    (libHandle   libHandleType);

Parameters

    libHandle                          A handle to the library to be unloaded.
Ora_Ffi.Unload_Library example

/* First load a dll library */
PROCEDURE register_libs IS
  test_lib  Ora_Ffi.Libhandletype;
BEGIN
  /* Load the testlib.dll library
  from directory C:\\libs\ */
  testlib_lhandle := ora_ffi.load_library
    (‘C:\\libs\’, ‘testlib.dll’);
END;

/* Generate PL/SQL Package containing
  functions from the test library. */
PROCEDURE define_c_funcs IS
  ...
  Ora_Ffi.Generate_Foreign (testlib_lhandle,
    ‘test_Ffi_Pkg’);
  ...
END;

/* Unload the library */
PROCEDURE unload_libs IS
BEGIN
  /* Unload the dll library assigned to the
  library handle ‘test_lib.’ */
  Ora_Ffi.Unload_library(testlib_lhandle);
  ...
END;

Ora_Ffi Example 1A

Suppose you want to create an interface to the C function pow, which is found in the
Microsoft Windows 95 runtime library: C:\windows\system\msvcrtd.dll. (The pow
function raises x to the y power.)

int pow(int x, int y)

First, create a package specification that represents the library and defines the PL/SQL
function that you want to invoke:

PACKAGE mathlib IS
  FUNCTION pow(x NUMBER, y NUMBER) RETURN NUMBER;
END;

You would call the PL/SQL function mathlib.pow, defined above, to invoke the foreign
function pow, from the dynamic library msvcrtd.dll.

Notice that this subprogram does not require a handle to the library or foreign function.
For convenience, the various registrations are handled in the package body, defined
below.
PACKAGE BODY mathlib IS
/* Declare the library and function handles. */
mathlib_lhandle  Ora_Ffi.Libhandletype;
to_power_fhandle Ora_Ffi.Funchandletype;

/* Create the PL/SQL function that will actually */
/* invoke the foreign function. */
FUNCTION ff_to_power(fhandle Ora_Ffi.Funchandletype,
x NUMBER, y NUMBER)RETURN NUMBER;
PRAGMA interface(C, ff_to_power, 11265);

/* Create the PL/SQL function that is defined in */
/* the package spec. This function simply */
/* passes along the arguments it receives to */
/* foreign function handle to the argument List. */
FUNCTION pow(x NUMBER, y NUMBER) RETURN NUMBER IS
BEGIN
RETURN(ff_to_power(to_power_fhandle, x, y));
END pow;

/* Define the body of package mathlib */
BEGIN

/* Load the library. */
mathlib_lhandle := Ora_Ffi.Load.Library
('C:\WINDOWS\SYSTEM\', 'msvcrt.dll');

/* Register the foreign function. */

/* Register both parameters of function to_power. */
Ora_Ffi.Register_Parameter(to_power_fhandle,
Ora_Ffi.C_DOUBLE);

/* Register the return type. */
Ora_Ffi.Register_Return(to_power_fhandle, Ora_Ffi.C_DOUBLE);
END; /* Package Body Mathlib */

To invoke the C function pow from msvcrtd.dll, you simply call the PL/SQL function pow, defined in the mathlib package specification. For example:

PL/SQL>
PROCEDURE raise_to_power (a in number, b in number) IS
BEGIN

110   Forms Developer and Reports Developer Built-in Package Reference
Ora_Ffi Example 1B

Here is an alternative way to implement the C function `pow`, shown in Ora_Ffi Example 1A. This example uses the Ora_Ffi.Generate_Foreign function to generate a PL/SQL package. The `PRAGMA` compiler directive, necessary to compile the foreign C function, is automatically included in the generated package, so it is not used in the package body below.

```sql
/* Create package mathlib that will generate a PL/SQL package using a foreign file C function to raise a number to a power. The parameter, pkg_name, lets you specify the name of the generated package. */
PACKAGE mathgen IS
  PROCEDURE gen(pkg_name IN VARCHAR2);
END;

PACKAGE BODY mathgen IS
  /* Define the 'gen' procedure that will generate the foreign file package. */
  PROCEDURE gen(pkg_name IN VARCHAR2) IS
    /* Declare the library and function handles. */
    mathlib_lhandle   Ora_Ffi.Libhandletype;
    to_power_fhandle  Ora_Ffi.Funchandletype;
    BEGIN   /* package body mathlib */
      /* Load the library. */
      mathlib_lhandle := Ora_Ffi.Load_Library
        ('C:\WINDOWS\SYSTEM\', 'msvcrt.dll');
      /* Register the foreign function. */
      to_power_fhandle := Ora_Ffi.Register_Function
        (mathlib_lhandle, 'pow', Ora_Ffi.C_Std);
      /* Register both parameters of the foreign function. */
      Ora_Ffi.Register_Parameter (to_power_fhandle, Ora_Ffi.C_DOUBLE);
      Ora_Ffi.Register_Parameter (to_power_fhandle, Ora_Ffi.C_DOUBLE);
      /* Register the return type of the foreign function. */
      Ora_Ffi.Register_Return (to_power_fhandle, Ora_Ffi.C_DOUBLE);
      /* Generate a PL/SQL package containing the foreign C function, 'pow.' You can name the new package by specifying a value for the parameter, pkg_name, when you generate the package. */
      Ora_Ffi.generate_foreign(mathlib_lhandle, pkg_name);
    END; /* Procedure gen */
END; /* Package Body mathgen */
```
To raise a number to a power with this method, you must first generate a PL/SQL package using package *mathgen* and procedure *gen*. For example, if the generated PL/SQL power package is called *mathlib*, you would generate it as follows:

```
PL/SQL> mathgen.gen('mathlib');
```

Then, to invoke the power function from package *mathlib*, you might write a procedure such as:

```
PROCEDURE raise_to_power (a in number, b in number) IS
    BEGIN
        text_io.put_line(mathlib.pow(a,b));
    END;
PL/SQL> raise_to_power(5,2);
25
```

### Ora_Ffi Example 2

Suppose you want to create an interface to the following C functions, which are located in the library C:\oralibs\imglib.dll:

```
void *get_image(char *imgkey)
void show_image(void *binimage, float iscale)
```

Assume that the function *get_image* uses a keyword argument to load image data, and then returns a generic pointer (i.e., a pointer of unspecified type) to that binary data. You then pass the pointer and a scaling factor to *show_image*, which displays the image on the screen.

First, create a package specification that represents the library and defines the PL/SQL functions that you want to invoke:

```
PACKAGE imglib IS
    FUNCTION get_image(ikey IN OUT VARCHAR2)
    RETURN Ora_Ffi.pointerType;

    PROCEDURE show_image(idata Ora_Ffi.pointerType,
            iscale NUMBER);
END; /* package imglib */
```

The package body is defined below:

```
PACKAGE BODY imglib IS
    /* Declare the library and function handles. */
    imglib_lhandle Ora_Ffi.libHandleType;
    get_image_fhandle Ora_Ffi.funcHandleType;
    show_image_fhandle Ora_Ffi.funcHandleType;

    /* Create the PL/SQL function that will actually */
    /* invoke the 'get_image' foreign function. */
    FUNCTION ff_get_image(fhandle Ora_Ffi.funcHandleType,
            ikey IN OUT VARCHAR2)
    RETURN Ora_Ffi.handleType;
    PRAGMA interface(C, ff_get_image, 11265);
```

---

112 Forms Developer and Reports Developer Built-in Package Reference
FUNCTION get_image(ikey IN OUT VARCHAR2) RETURN Ora_Ffi.pointerType IS
    ptr Ora_Ffi.pointerType;
BEGIN
    ptr.handle := ff_get_image(get_image_fhandle, ikey);
    RETURN(ptr);
END; /* function get_image */

PROCEDURE ff_show_image(fhandle Ora_Ffi.funcHandleType,
    idata Ora_Ffi.handleType,
    iscale NUMBER);
PRAGMA interface(C, ff_show_image, 11265);

PROCEDURE show_image(idata Ora_Ffi.pointerType,
    iscale NUMBER) IS
BEGIN
    ff_show_image(show_image_fhandle, idata.handle, iscale);
END; /* procedure show_image */

BEGIN   /* package body imglib */
/* Load the library. */
imglib_lhandle := Ora_Ffi.Load_Library
    ('C:\oralibs\', 'imglib.dll');

/* Register the foreign functions. */
get_image_fhandle := Ora_Ffi.Register_Function
    (imglib_lhandle, 'get_image', Ora_Ffi.C_Std);
show_image_fhandle := Ora_Ffi.Register_Function
    (imglib_lhandle, 'show_image', Ora_Ffi.C_Std);

/* Register the parameters. */
Ora_Ffi.Register_Parameter(get_image_fhandle,
    Ora_Ffi.C_Char_Ptr);
Ora_Ffi.Register_Parameter(show_image_fhandle,
    Ora_Ffi.C_Void_Ptr);
Ora_Ffi.Register_Parameter(show_image_fhandle,
    Ora_Ffi.C_Float);

/* Register the return type (’get_image’ only). */
Ora_Ffi.Register_Return(get_image_fhandle,
    Ora_Ffi.C_Void_Ptr);

END; /* package body imglib */

To invoke the foreign functions, you would call the PL/SQL procedures defined in the package specification, as in the following example:

PROCEDURE display_image(keywrd IN OUT VARCHAR2) IS
    img_ptr Ora_Ffi.PointerType;
BEGIN
    img_ptr := imglib.get_image(keywrd);
imglib.show_image(img_ptr, 2);
END; /* procedure display_image */
Ora_NLS Package

Ora_NLS package

Ora_Nls.American
Ora_Nls.American_Date
Ora_Nls.Bad_Attribute
Ora_Nls.Get_Lang_Scalar
Ora_Nls.Get_Lang_Str
Ora_Nls.Linguistic_Collate
Ora_Nls.Linguistic_Specials
Ora_Nls.Modified_Date_Fmt
Ora_Nls.No_Item
Ora_Nls.Not_Found
Ora_Nls.Right_to_Left
Ora_Nls.Simple_Cs
Ora_Nls.Single_Byte

Ora_Nls.American

**Description**  Returns TRUE or FALSE, depending on whether the current character set is "American".

**Syntax**

```
FUNCTION Ora_Nls.American
RETURN BOOLEAN;
```

**Returns**  TRUE or FALSE.

**Ora_Nls.American example**

```
/*
 ** Determine if you’re dealing with an American
 ** set or not
 */
PROCEDURE is_american (out Text_IO.File_Type) IS
us  BOOLEAN;
BEGIN
  us := Ora_Nls.American;
  IF us = TRUE
    Text_IO.Put (out, 'Character set it American');
  ELSE
    change_char_set;
  ENDIF
END;

Ora_Nls.American_Date

**Description**  Returns TRUE or FALSE, depending on whether the current date format is "American".

**Syntax**
FUNCTION Ora_Nls.American_Date
RETURN BOOLEAN;

**Returns**  TRUE or FALSE.

**Ora_Nls.American_Date example**

```/*
  ** Determine if date format is American
*/
PROCEDURE is_amerdate (out Text_IO.File_Type) IS
  usd  BOOLEAN;
BEGIN
  usd := Ora_Nls.American_Date;
  IF usd = TRUE
    Text_IO.Put (out, 'Date format it American');
  ELSE
    change_date_to_us;
  ENDIF
END;
```

Ora_Nls.Bad_Attribute

**Description**  Raised when no attribute is supplied to Ora_Nls.Get_Lang_Scalar or Ora_Nls.Get_Lang_Str.

**Syntax**
Ora_Nls.Bad_Attribute  EXCEPTION;
Ora_Nls.Bad_Attribute example

/*
  ** Handle the Bad Attribute exception
  */
EXCEPTION
  WHEN Ora_Nls.Bad_Attribute THEN
    Text_IO.Put_Line('Check calls to Get_Lang_Scalar
    and Get_Lang_Str.  A bad attribute name was found.');

Ora_Nls.Get_Lang_Scalar

Description  Returns the requested information about the current language. You can use
GET_LANG_SCALAR to retrieve numeric data.

Syntax
FUNCTION Ora_Nls.Get_Lang_Scalar
  (attribute PLS_INTEGER)
RETURN NUMBER;

Parameters
attribute  An Ora_Nls Constant or its associated integer
value. For a List Of constants, see Ora_Nls
Constants.

Returns  A number.

Ora_Nls.Get_Lang_Scalar example

/*
  ** Retrieve and print out the language number
  */
BEGIN
  lang_num := Ora_Nls.Get_Lang_Scalar
  (Ora_Nls.Iso_Alphabet);
  Text_IO.Putf (out, "Current Language numer is %s\n",
    lang_num);
END;

Ora_Nls.Get_Lang_Str

Description  Returns the requested information about the current language. You can use
GET_LANG_STR to retrieve character information.

Syntax
FUNCTION Ora_Nls.Get_Lang_Str
   (attribute PLS_INTEGER)
RETURN VARCHAR2;

Parameters
   attribute An Ora_Nls Constant or its associated integer value. For a List Of constants, see Ora_Nls Constants.

Returns A character value.

Ora_Nls.Get_Lang_Str example

/*
 ** Retrieve and print out the language name
*/
BEGIN lang_name (out Text_IO.File_Type)
   lang_name VARCHAR2(80);
BEGIN
   lang_name := Ora_Nls.Get_Lang_Str
      (Ora_Nls.Language);
   Text_IO.Putf (out, "Current Language is %s\n",
      lang_name);
END;

Ora_Nls.Linguistic_Collate

Description Returns TRUE or FALSE, depending on whether the characters in the current character set need to be collated according to special linguistic information.

Syntax
FUNCTION Ora_Nls.Linguistic_Collate
RETURN BOOLEAN;

Returns TRUE or FALSE.

Usage Notes If this function returns TRUE, a binary sort of two characters will not necessarily return the correct value. This is because encoding schemes for character sets do not necessarily define all characters in ascending numerical order.
In addition, the sort position of a character may vary for different languages. For example, an "ä" is sorted before "b" in German, but after "z" in Swedish.

Ora_Nls.Linguistic_Collate example

/*
 ** Determine whether or not special collating is needed.
 */
collate := Ora_Nls.Linguistic_Collate;
IF collate = TRUE THEN
   lang_name (langinfo.txt);
   Text_IO.Put ('This needs special collating.');
**Ora_Nls.Linguistic_Specials**

**Description**  Returns true or false, depending on whether there are linguistic specials in use.

**Syntax**
FUNCTION Ora_Nls.Linguistic_Specials
RETURN BOOLEAN;

**Returns**  TRUE or FALSE.

**Usage Notes**  Linguistic specials are language-specific special cases for collation and case conversion (upper and lower). An example: The uppercase for the German sharp "s" (one byte), which is "SS" (two bytes). Sorting: Also done according to the two-byte value. Linguistic specials are defined in a linguistic definition along with normal collation. When there are linguistic specials defined for the linguistic definition that is in effect for a specific language handle, output sizes of functions handling linguistic specials can be larger than input string sizes.

**Ora_Nls.Linguistic_Specials example**

```/*
** Determine whether or not specials are in use
** and how to deal with them if so
*/
specials := Ora_Nls.Linguistic_Specials;
IF specials = TRUE THEN
    lang_name (langinfo.txt);
    Text_IO.Put ('Specials are in use.');
ENDIF;
```

**Ora_Nls.Modified_Date_Fmt**

**Description**  Returns true or false, depending on whether the date format has been modified.

**Syntax**
FUNCTION Ora_Nls.Modified_Date_Fmt
RETURN BOOLEAN;

**Returns**  TRUE or FALSE.

**Ora_Nls.Modified_Date_Fmt example**

```/*
 */
```

---

Forms Developer and Reports Developer Built-in Package Reference  119
** Determine whether or not the date format has been modified
*/
modify := Ora_Nls.Modified_DateFmt;
IF modify = TRUE
   Text_IO.Putf (langinfo.txt, 'The date format has been modified.');
ENDIF;

---

### Ora_Nls.No_Item

**Description** Raised when a user-supplied attribute cannot be located in the List Of attributes constants.

**Syntax**

```
Ora_Nls.No_Item EXCEPTION;
```

**Ora_Nls.No_Item example**

```
/*
 ** Hand the exception for an unidentified attribute constant
 */
EXCEPTION
   WHEN Ora.Nls.No_Item THEN
      Text_IO.Put ('An attribute supplied is not valid.');
```

---

### Ora_Nls.Not_Found

**Description** This exception is raised when a requested item cannot be found. This is most likely caused by using Ora_Nls.Get_Lang_Scalar to retrieve character information, or by using Ora_Nls.Get_Lang_Str to retrieve numeric information.

**Syntax**

```
Ora_Nls.Not_Found EXCEPTION;
```

**Ora_Nls.Not_Found example**

```
/*
 ** Hand the exception for an item that was not found
 */
EXCEPTION
   WHEN Ora.Nls.Not_Found THEN
      Text_IO.Put ('The item was not found, check calls to Get_Lang.');
```
Ora_Nls.Right_to_Left

Description  Returns true or false, depending on whether the writing direction of the current language is "right-to-left".

Syntax
FUNCTION Ora_Nls.Right_To_Left
RETURN BOOLEAN;

Returns  TRUE or FALSE.

Ora_Nls.Right_To_Left example

/*
 ** Verify that the language is a right-to-left language
*/
rtl := Ora_Nls.Right_To_Left;
IF rtl = FALSE
  Text_IO.Put (langinfo.txt, 'This is not a right to left language.');
ENDIF;

Ora_Nls.Simple_Cs

Description  Returns true or false, depending on whether the current character set is simple (i.e., single-byte, no special characters, no special handling).

Syntax
FUNCTION Ora_Nls.Simple_Cs
RETURN BOOLEAN;

Returns  TRUE or FALSE.

Ora_Nls.Simple_Cs example

/*
 ** Determine if the language is simple or not
*/
simplecs := Ora_Nls.Simple_Cs;
IF simplecs = TRUE
  lang_name (langinfo.txt);
  Text_IO.Put ('This language uses a simple character set.');
ELSE
  lang_name (langinfo.txt);
  Text_IO.Put ('This language uses a complex character set.');
ENDIF;
**Ora_Nls.Single_Byte**

**Description** Returns true or false, depending on whether all of the characters in the current character set can be represented in one byte.

**Syntax**
```plsql
FUNCTION Ora_Nls.Single_Byte
RETURN BOOLEAN;
```

**Returns** TRUE or FALSE.

**Ora_Nls.Single_Byte example**
```plsql
/*
** Determine if the character set is single or multi-byte
*/
bytes := Ora_Nls.Single_Byte;
IF bytes = FALSE
   lang_name (langinfo.txt);
   Text_IO.Put ('This is a multi-byte character set.');
END IF;
```
Ora_Prof Package

Ora_Prof package

Ora_Prof.Bad_Timer
Ora_Prof.Create_Timer
Ora_Prof.Destroy_Timer
Ora_Prof.Elapsed_Time
Ora_Prof.Reset_Timer
Ora_Prof.Start_Timer
Ora_Prof.Stop_Timer

Ora_Prof.Bad_Timer

Description  Raised when an invalid timer name is supplied to another Ora_Prof package procedure or function.

Syntax

Ora_Prof.Bad_Timer  EXCEPTION;

Ora_Prof.Bad_Timer example

/*
 ** Create a timer, start it, run a subprogram,
 ** stop the timer, then display the time in
 ** seconds. Destroy the timer when finished.
 */
PROCEDURE timed_proc (test VARCHAR2) IS
  i PLS_INTEGER;
BEGIN
  Ora_Prof.Create_Timer('loop2');
  Ora_Prof.Start_Timer('loop2');
  test;
  Ora_Prof.Stop_Timer('loop2');
  Text_IO.Putf('Loop executed in %s seconds.\n',
    Ora_Prof.Elapsed_Time('loop2'));
  Ora_Prof.Destroy_Timer('loop2');
EXCEPTION
WHEN ORA_PROF.BAD_TIMER THEN
  text_io.put_line('Invalid timer name');
END;

Ora_Prof.Create_Timer

Description  Allocates the named timer. Any references to the named timer before this service is used will raise an error.

Syntax
PROCEDURE Ora_Prof.Create_Timer
  (timer VARCHAR2);

Parameters
  timer  The name of the timer.

Ora_Prof.Create_Timer example

  /*
   **Allocate the timer 'LOOPTIME'.
   */
  Ora_Prof.Create_Timer('LOOPTIME');

Ora_Prof.Destroy_Timer

Description  Destroys the named timer. All memory associated with the timer is freed at that time. Any references to the named timer after this service is used will raise an error.

Syntax
PROCEDURE Ora_Prof.Destroy_Timer
  (timer VARCHAR2);

Parameters
  timer  The name of the timer.

Ora_Prof.Destroy_Timer example

  /*
   **Destroy the timer 'LOOPTIME'.
   */
  Ora_Prof.Destroy_Timer('LOOPTIME');
Ora_Prof.Elapsed_Time

Description  Returns the amount of time accumulated in the code timer since the last call to Ora_Prof.Reset_Timer.

Syntax
FUNCTION Ora_Prof.Elapsed_Time
  (timer  PLS_INTEGER)
RETURN PLS_INTEGER;

Parameters
  timer  The name of the timer.

Returns  The amount of time (in milliseconds) accumulated in the code timer.

Ora_Prof.Elapsed_Timer example

/*
 ** Create a timer, start it, run a subprogram,
 ** stop the timer, then display the time in seconds. Destroy the timer when finished.
 */
PROCEDURE timed_proc (test VARCHAR2) IS
  i PLS_INTEGER;
BEGIN
  Ora_Prof.Create_Timer('loop2');
  Ora_Prof.Start_Timer('loop2');
  test;
  Ora_Prof.Stop_Timer('loop2');
  Text_IO.Putf('Loop executed in %s seconds.
',
    Ora_Prof.Elapsed_Time('loop2'));
  Ora_Prof.Destroy_Timer('loop2');
END;

Ora_Prof.Reset_Timer

Description  Resets the elapsed time of a timer to zero.

Syntax
PROCEDURE Ora_Prof.Reset_Timer
  (timer VARCHAR2);

Parameters
  timer  The name of the timer.

Ora_Prof.Reset_Timer example

PROCEDURE multi_time IS
  i PLS_INTEGER;

BEGIN
  Ora_Prof.Create_Timer('loop');
  --
  -- First loop...
  --
  Ora_Prof.Start_Timer('loop');
  FOR i IN 1..10 LOOP
    Text_IO.Put_Line('Hello');
    END LOOP;
  Ora_Prof.Stop_Timer('loop');
  --
  -- Second loop...
  --
  Ora_Prof.Start_Timer('loop');
  FOR i IN 1..10 LOOP
    Text_IO.Put_Line('Hello');
    END LOOP;
  Ora_Prof.Stop_Timer('loop');
  Ora_Prof.Destroy_Timer('loop');
END;

Ora_Prof.Start_Timer

Description  Starts a timer. Any time accumulated between calls to Ora_Prof.Timer_Start and Ora_Prof.Timer_Stop added to the timer’s total elapsed time.

Syntax
PROCEDURE Ora_Prof.Start_Timer
  (timer VARCHAR2);

Parameters

  timer The name of the timer.

Ora_Prof.Start_Timer example

PROCEDURE multi_time IS
  i PLS_INTEGER;
BEGIN
  Ora_Prof.Create_Timer('loop');
  --
  -- First loop...
  --
  Ora_Prof.Start_Timer('loop');
  FOR i IN 1..10 LOOP
    Text_IO.Put_Line('Hello');
    END LOOP;
  Ora_Prof.Stop_Timer('loop');
  --
  -- Second loop...
  --
  Ora_Prof.Start_Timer('loop');
  FOR i IN 1..10 LOOP

Forms Developer and Reports Developer Built-in Package Reference
Text_IO.Put_Line('Hello');
END LOOP;
Ora_Prof.Stop_Timer('loop');
Ora_Prof.Destroy_Timer('loop');
END;

Ora_Prof.Stop_Timer

**Description**  Stops a timer. Any time accumulated between calls to Ora_Prof.Timer_Start and Ora_Prof.Timer_Stop added to the timer’s total elapsed time.

**Syntax**

```
PROCEDURE Ora_Prof.Stop_Timer
  (timer VARCHAR2);
```

**Parameters**

- `timer`  The name of the timer.

**Ora_Prof.Stop_Timer example**

```
PROCEDURE multi_time IS
  i PLS_INTEGER;
BEGIN
  Ora_Prof.Create_Timer('loop');
  --
  -- First loop...
  --
  Ora_Prof.Start_Timer('loop');
  FOR i IN 1..10 LOOP
    Text_IO.Put_Line('Hello');
    END LOOP;
  Ora_Prof.Stop_Timer('loop');
  --
  -- Second loop...
  --
  Ora_Prof.Start_Timer('loop');
  FOR i IN 1..10 LOOP
    Text_IO.Put_Line('Hello');
    END LOOP;
  Ora_Prof.Stop_Timer('loop');
  Ora_Prof.Destroy_Timer('loop');
END;
```
Text_IO Package

Text_IO package

Text_IO.Fclose
Text_IO.File_Type
Text_IO.Fopen
Text_IO.Is_Open
Text_IO.Get_Line
Text_IO.New_Line
Text_IO.Put
Text_IO.Putf
Text_IO.Put_Line

Text_IO.Fclose

Description  Closes an open file.
Syntax
PROCEDURE Text_IO.Fclose
  (file file_type);
Parameters

  file  A variable that specifies the file to close.

Text_IO.Fclose example

  /*
   ** Close the output file.
   */
  Text_IO.Fclose (out_file);
Text_IO.File_Type

Description  Specifies a handle to a file.

Syntax

```plaintext
TYPE Text_IO.File_Type;
```

Text_IO.File_Type example

```plaintext
/*
** Declare a local variable to represent
** the output file you will write to.
*/
out_file  Text_IO.File_Type;
```

Text_IO.Fopen

Description  Opens the designated file in the specified mode.

Syntax

```plaintext
FUNCTION Text_IO.Fopen
  (spec      VARCHAR2,
   filemode  VARCHAR2)
RETURN Text_IO.File_Type;
```

Parameters

- `spec` A case-insensitive string corresponding to a file’s name.
- `filemode` A single case-insensitive character that specifies the mode in which to open the file, and consists of one of the following characters:
  - R  Open the file for reading only.
  - W  Open the file for reading and writing after deleting all existing lines in the file.
  - A  Open the file for reading and writing without deleting existing lines (i.e., appending).

Returns  A handle to the specified file.

Text_IO.Fopen example

```plaintext
/*
** Declare two local variables to represent two files:
** one to read from, the other to write to.
*/
in_file   Text_IO.File_Type;
```
out_file  := Text_IO.Fopen('salary.txt', 'r');
out_file := Text_IO.Fopen('bonus.txt', 'w');

---

### Text_IO.Is_Open

**Description** Checks to see if the specified file is currently open.

**Syntax**

```plaintext
FUNCTION Text_IO.Is_Open
  (file  file_type)
RETURN BOOLEAN;
```

**Parameters**

- `file` A variable that specifies the file to check.

**Returns** TRUE or FALSE.

**Text_IO.Is_Open example**

```plaintext
/*
 ** Determine if the output file is open. If so, then close it.
 */
IF Text_IO.Is_Open(out_file) THEN
  Text_IO.Fclose(out_file);
```

---

### Text_IO.Get_Line

**Description** Retrieves the next line of an open file and places it in `item`.

Text_IO.Get_Line reads characters until a newline character (i.e., carriage return) is read or an end-of-file (EOF) condition is encountered.

If the line to be read exceeds the size of `item`, the Value_Error exception is raised. If there are no more characters remaining in the file, the No_Data_Found exception is raised.

**Syntax**

```plaintext
PROCEDURE Text_IO.Get_Line
  (file  file_type,
   item  OUT VARCHAR2);
```

**Parameters**

- `file` A variable that specifies an open file.
- `item` A variable used to hold the next line read
Text_IO.Get_Line example

/*
 ** Open a file and read the first line
 ** into linebuf.
 */
in_file  Text_IO.File_Type;
linebuf  VARCHAR2(80);
in_file := Text_IO.Fopen('salary.txt, 'r');
Text_IO.Get_Line(in_file,linebuf);

Text_IO.New_Line

Description  Concatenates the specified number of newline characters (i.e., carriage returns) to the current line of an open file, or outputs them to the Interpreter. The default is 1, that is, if you specify no number (e.g., Text_IO.New_Line;) a single newline character is created.

Syntax
PROCEDURE Text_IO.New_Line
(file file_type,
 n PLS_INTEGER := 1);
PROCEDURE Text_IO.New_Line
(n PLS_INTEGER := 1);

Parameters

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>A variable that specifies an open file.</td>
</tr>
<tr>
<td>n</td>
<td>An integer.</td>
</tr>
</tbody>
</table>
Text_IO.New_Line example

/*
** Write a string to the output file, then
** create a newline after it.
*/
Text_IO.Put(out_file, SYSDATE);
Text_IO.New_Line(out_file, 2);

Text_IO.Put

Description  Concatenates the supplied data to the current line of an open file, or outputs it to the Interpreter. Notice that there are several Text_IO.Put procedures, which accept VARCHAR2, DATE, NUMBER, and PLS_INTEGER values for item. All of the procedures (except VARCHAR2) convert the supplied data to a character string. No newline character (i.e., carriage return) Added.

Syntax
PROCEDURE Text_IO.Put
  (file  file_type,
   item VARCHAR2);
PROCEDURE Text_IO.Put
  (item VARCHAR2);
PROCEDURE Text_IO.Put
  (item DATE);
PROCEDURE Text_IO.Put
  (file  file_type,
   item DATE);
PROCEDURE Text_IO.Put
  (file  file_type,
   item NUMBER);
PROCEDURE Text_IO.Put
  (item NUMBER);
PROCEDURE Text_IO.Put
  (file  file_type,
   item PLS_INTEGER);
PROCEDURE Text_IO.Put
  (item PLS_INTEGER);

Parameters

  file  A variable that specifies an open file.
  item  A variable to be used as a buffer.
Text_IO.Put example

/*
** Write a line to a specified output file, create
** a newline, then write another line to the output
** file.
*/
Text_IO.Put(out_file, SYSDATE);
Text_IO.New_Line(out_file);
Text_IO.Put('Processing ends...');

Text_IO.Putf

Description Formats and writes a message to an open file, or outputs the message to the Interpreter. You can embed up to five "%s" patterns within format (e.g., "%s %s %s"). The "%s" patterns are replaced with successive character arg values (e.g., 'Check', 'each', 'value.'). "\n" patterns are replaced with newline characters (i.e., carriage returns).

Syntax
PROCEDURE Text_IO.Putf
    (arg    VARCHAR2);
PROCEDURE Text_IO.Putf
    (file   file_type,
     arg    VARCHAR2);
PROCEDURE Text_IO.Putf
    (file   file_type,
     format  VARCHAR2,
     [arg1  [,..., arg5] VARCHAR2]);
PROCEDURE Text_IO.Putf
    (format VARCHAR2,
     [arg1  [,..., arg5] VARCHAR2]);

Parameters
arg An argument that specifies the value to be displayed (e.g., character string, variable).
format Specifies the format of the message to be displayed.
file A variable that specifies an open file.

Usage Notes To format messages containing non-character substitutions, use the TO_CHAR function on the argument (see the example below).

Text_IO.Putf example

/*
** Write a line to the output file, using the
** TO_CHAR(SYSDATE) call to represent the substituted
** character variable.
*/

134 Forms Developer and Reports Developer Built-in Package Reference
Text_IO.Put_Line

**Description**  Concatenates the character data supplied by `item` to the current line of an open file, or outputs it to the Interpreter. A newline character (i.e., carriage return) is automatically added to the end of the string.

**Syntax**

```plaintext
PROCEDURE Text_IO.Put_Line
  (file file_type,
   item VARCHAR2);
```

**Parameters**

- `file`  A variable that specifies an open file.
- `item`  A variable that specifies the character data to be displayed.

**Text_IO.Put_Line example**

```plaintext
/*
** Print two complete lines to the output file.
*/
Text_IO.Put_Line(out_file, TO_CHAR(SYSDATE));
Text_IO.Put_Line('Starting test procedures...');
*/
```
Tool_Eng Package

Tool_Eng package

Tool_Eng.Getvar

Description Provides a way to import an environment variable into a VARCHAR2 variable.

Syntax
PROCEDURE Tool_Eng.Getvar
  (varname VARCHAR2,
   varvalue VARCHAR2);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>The name of the environment variable.</td>
</tr>
<tr>
<td>varvalue</td>
<td>The value of the environment variable.</td>
</tr>
</tbody>
</table>
**Tool.Env.Getvar example**

```
/*
** Retrieve the environment variable USER into a
** variable named :userid so you can use it in a
** connect string or other call.
*/
Tool.Env.Getvar('USER', :userid);
```
Tool_Err Package

Tool_Err package

Tool_Err.Clear
Tool_Err.Code
Tool_Err.Encode
Tool_Err.Message
Tool_Err.Nerrors
Tool_Err.Pop
Tool_Err.Tool_Error
Tool_Err.Toperror

Tool_Err.Clear

Description Discards all errors currently on the error stack.
Syntax
PROCEDURE Tool_Err.Clear;

Tool_Err.Code

Description Returns the error code for the ith error on the error stack (the default is the top-most error). If there are no errors on the stack, zero is returned.
Syntax
FUNCTION Tool_Err.Code
(i PLS_INTEGER := TOPERROR)
RETURN NUMBER;
Parameters
i An integer that specifies an error on the error stack.
Returns The error code of the error specified.
/*
** Check for unexpected error, disregard it, and print any other error.
*/
PROCEDURE check_err IS
BEGIN
  IF (TOOL_ERR.CODE != pkg_a.not_found) THEN
    TEXT_IO.PUT_LINE(TOOL_ERR.MESSAGE);
  END IF;
  TOOL_ERR.POP;
END;

Tool_Err.Encode

Description  Given a prefix and an offset, constructs a unique error code for use within a package.
Note: This is not a PL/SQL exception.
Syntax
FUNCTION Tool_Err.Encode
  (prefix VARCHAR2,
   offset PLS_INTEGER)
RETURN NUMBER;
Parameters
  prefix A string of five characters.
  offset An integer from 1 to 127.
Returns  An error code.

Tool_Err.Encode example

/*
** Define a list of errors for a package called pkg_a.
*/
PACKAGE pkg_a IS
  not_found CONSTANT pls_integer := TOOL_ERR.ENCODE('pkg_a', 1);
  bad_value CONSTANT pls_integer := TOOL_ERR.ENCODE('pkg_a', 2);
  too_big   CONSTANT pls_integer := TOOL_ERR.ENCODE('pkg_a', 3);
  too_small CONSTANT pls_integer := TOOL_ERR.ENCODE('pkg_a', 4);
  . . .     /* Rest of pkg_a specification */
END;
Tool_Err.Message

Description  Returns the formatted message associated with the \( i \)th error on the error stack (the default is the top-most error).

Syntax
FUNCTION Tool_Err.Message
  (i PLS_INTEGER := TOPERROR)
RETURN VARCHAR2;

Parameters
  \( i \)  An integer that specifies an error on the error stack.

Returns  An error message.

Tool_Err.Message example

```
/*
** Determine the number of errors
** on the stack. Then, loop through stack,
** and print out each error message.
*/
PROCEDURE print_all_errors IS
    number_of_errors   PLS_INTEGER;
BEGIN
    EXCEPTION
        WHEN OTHERS THEN
            number_of_errors := TOOL_ERR.NERRORS;
            FOR i IN 1..number_of_errors LOOP
                TEXT_IO.PUT_LINE(TOOL_ERR.MESSAGE(i-1));
            END LOOP;
    END;
END;
```

Tool_Err.Nerrors

Description  Returns the number of errors currently on the error stack.

Syntax
FUNCTION Tool_Err.Nerrors
RETURN PLS_INTEGER;

Returns  The number of error on the error stack.

Tool_Err.Nerrors example

```
/*
** Determine the number of errors
** on the stack. Then, loop through stack,
** and print out each error message.
*/
PROCEDURE print_all_errors IS
```
number_of_errors   PLS_INTEGER;
BEGIN
EXCEPTION
WHEN OTHERS THEN
    number_of_errors := TOOL_ERR.NERRORS;
FOR i IN 1..number_of_errors LOOP
    TEXT_IO.PUT_LINE(TOOL_ERR.MESSAGE(i-1));
END LOOP;
END;

---------

Tool_Err.Pop

**Description**  Discards the top-most error on the error stack.

**Syntax**

PROCEDURE Tool_Err.Pop;

**Tool_Err.Pop example**

/*
** Loop through each message in the stack,
** print it, then clear the top most error.
*/
BEGIN
    . . .
EXCEPTION
    WHEN OTHERS THEN
    FOR i IN 1..Tool_Err.Nerrors LOOP
        TEXT_IO.PUT_LINE(TOOL_ERR.MESSAGE);
        TOOL_ERR.POP;
    END LOOP;
    . . .
END;

---------

Tool_Err.Tool_Error

**Description**  Defines a generic error you can raise to indicate that one or more errors have been pushed onto the error stack.

**Syntax**

Tool_Err.Tool_Error   EXCEPTION;

**Tool_Err.Tool_Error example**

/*
** Raise a generic internal error if a function argument is out of range. */
PROCEDURE my_proc(count PLS_INTEGER) IS
BEGIN
  IF (count < 0) THEN
    RAISE TOOL_ERR.TOOL_ERROR;
  END IF;
  . . .
END;

---

**Tool_Err.Toperror**

**Description** Identifies the top-most error on the error stack.

**Syntax**

```plsql
Tool_Err.Toperror CONSTANT PLS_INTEGER;
```

**Tool_Err.Toperror example**

```plsql
/*
** Print top-most error on the stack. The same results are produced by calling Tool_Err.Message
** with no arguments.
*/
BEGIN
  . . .
  TEXT_IO.PUT_LINE(TOOL_ERR.MESSAGE(TOOL_ERR.TOPERROR));
  . . .
END;
```
Tool_Res Package

Tool_Res package
Tool_Res.Bad_File_Handle
Tool_Res.Buffer_Overflow
Tool_Res.File_Not_Found
Tool_Res.No_Resource
Tool_Res.Rfclose
Tool_Res.Rfhandle
Tool_Res.Rfopen
Tool_Res.Rfread

Tool_Res.Bad_File_Handle
Description Raised when the file handle passed to Tool_Res.Rfclose is invalid.
Syntax
Tool_Res.Bad_File_Handle EXCEPTION;

Tool_Res.Bad_File_Handle example
/*
** This examples uses Tool_Res.Bad_File_Handle
*/
PROCEDURE res_test IS
resfileh TOOL_RES.RFHANDLE;
resfileh1 TOOL_RES.RFHANDLE;
resl VARCHAR2(20);
BEGIN
/* Open a resource file */
resfileh := TOOL_RES.RFOPEN('C:\resource\test.res');
...
/* Used wrong handle to close
the resource file. */
TOOL_RES.RFCLOSE(resfileh);
...
EXCEPTION
WHEN TOOL_RES.BAD_FILE_HANDLE THEN
    /* print error message */
    TEXT_IO.PUT_LINE('Invalid file handle.');
    /* discard the error */
    TOOL_ERR.POP;
END;

---

### Tool_Res.Buffer_Overflow

**Description**  Raised when you tried to get a resource that was longer than the supplied buffer.

**Syntax**

```
Tool_Res.Buffer_Overflow  EXCEPTION;
```

### Tool_Res.Buffer_Overflow example

```
/*
** This example uses Tool_Res.Buffer_Overflow
*/
PROCEDURE res_buf_test IS
    resfileh   TOOL_RES.RFHANDLE;
    res1       VARCHAR2(20);
BEGIN
    /*  Open a resource file */
    resfileh := TOOL_RES.RFOPEN
        ('C:\resource\test.res');
    /*  Attempt to read very large string
        which overflows buffer. */
    res1 := TOOL_RES.RFREAD(resfileh,'res_1');
    ...

EXCEPTION
    WHEN TOOL_RES.BUFFER_OVERFLOW THEN
        /* print error message */
        TEXT_IO.PUT_LINE('Buffer overflow.');
        /* discard the error */
        TOOL_ERR.POP;
END;
```

---

### Tool_Res.File_Not_Found

**Description**  Raised when the specified file cannot be opened, most likely because of one of the following reasons:

- file name
- permissions on the file
- system error

**Syntax**

```
Tool_Res.File_Not_Found  EXCEPTION;
```

**Tool_Res.File_Not_Found example**

```
/*
** This example uses Tool_Res.File_Not_Found
*/
PROCEDURE res_test IS
  resfileh   TOOL_RES.RFHANDLE;
  res1       VARCHAR2(20);
BEGIN
  /* Open a resource file */
  resfileh := TOOL_RES.RFOPEN
              ('C:\resource\twst.res');
  /* File name is misspelled. */
  ...
EXCEPTION
  WHEN TOOL_RES.FILE_NOT_FOUND THEN
    /* print error message */
    TEXT_IO.PUT_LINE('Cannot find the file.');
    /* discard the error */
    TOOL_ERR.POP;
END;
```

**Tool_Res.No_Resource**

**Description** This exception is raised when the named resource could not be found. If a file was specified, the resource does not exist in that file. If no file was specified, the resource does not exist in any of the resource files that are currently open.

**Syntax**

```
Tool_Res.No_Resource  EXCEPTION;
```

**Tool_Res.No_Resource example**

```
/*
** This examples uses Tool_Res.No_Resource
*/
PROCEDURE res_test IS
  resfileh   TOOL_RES.RFHANDLE;
  res1       VARCHAR2(20);
BEGIN
  /* Open a resource file */
  resfileh := TOOL_RES.RFOPEN
```
('C:\resource\test.res');
/* Attempt to read nonexistent
   resource from file. */
res1 := TOOL_RES.RFREAD(resfileh,'Res_1');
...

EXCEPTION
  WHEN TOOL_RES.NO_RESOURCE THEN
    /* print error message */
    TEXT_IO.PUT_LINE('Cannot find the resource. ');
    /* discard the error */
    TOOL_ERR.POP;
END;

Tool_Res.Rfclose

Description  Closes the specified resource file. All files opened with Tool_Res.Rfopen should be closed using Tool_Res.Rfclose before quitting the application.

Syntax
PROCEDURE Tool_Res.Rfclose
  (file rfhandle);

Parameters
  file  A file to close.

Usage Notes  The following exceptions may be raised by RFCLOSE:
BAD_FILE_HAN Raised if the file handle does not point to a
   valid file.
Tool_Err.Tool_Er Raised if an internal error is trapped.

Tool_Res.Rfclose example

/*
 ** This examples uses Tool_Res.Rfclose
 */
PROCEDURE my_cleanup
  (my_file_handle IN OUT TOOL_RES.RFHANDLE) IS
BEGIN
  /* Assign a resource file to 'fhandle.' */
  ...
  /* Close the resource file with the
     handle 'fhandle.' */
  TOOL_RES.RFCLOSE(my_file_handle);
  ...
END;
Tool_Res.Rfhandle

Description  Specifies a handle to a file.
Syntax
   TYPE Tool_Res.Rfhandle;

Tool_Res.Rfhandle example

   /* This examples uses Tool_Res.Rfhandle */
   PROCEDURE res_test IS
       resfileh   TOOL_RES.RFHANDLE;
       BEGIN
           /* Assign handle to a resource file */
           resfileh := TOOL_RES.RFOPEN('C:\test.res');
           ...
       END;

Tool_Res.Rfopen

Description  Opens the specified file as a resource file.
Syntax
   FUNCTION Tool_Res.Rfopen
       (spec VARCHAR2)
   RETURN rfhandle;
Parameters
   spec  A file to be opened. spec is not case-sensitive.
Returns  A handle to the specified file.
Usage Notes  The following exceptions may be raised by Tool_Res.Rfopen:
               File_Not_Found  Raised if spec does not point to a valid file, or
               the file cannot be opened.
               Tool_Err.Tool_Er Raised if an internal error is trapped.
               ...

Tool_Res.Rfopen example

   /*
      ** This example uses Tool_Res.Rfopen
   */
   PROCEDURE my_init
(fhandle OUT TOOL_RES.RFHANDLE) IS
BEGIN
/* Open a resource file and assign it to
the handle, 'fhandle.' */
fhandle := TOOL_RES.RFOPEN('C:\my_app.res');
...
END;

---

**Tool_Res.Rfread**

**Description**  Reads the specified resource. If a file handle is included, only the specified resource file will be searched for the named resource. Otherwise, all currently open resource files will be searched.

**Syntax**

```sql
FUNCTION Tool_Res.Rfread
  (rfile   rfhandle,
   resid   VARCHAR2,
   restype VARCHAR2 := 'string')
RETURN VARCHAR2;
FUNCTION Tool_Res.Rfread
  (resid   VARCHAR2,
   restype VARCHAR2 := 'string')
RETURN VARCHAR2;
```

**Parameters**

- **rfile**  A file to read.
- **resid**  A resource ID.
- **restype**  The type of resource.

**Returns**  A handle to the specified file.

**Usage Notes**  The following exceptions may be raised by Rfread:

- **No_Resource**  Raised if the named resource could not be located.
- **Buffer_Overflow**  Raised if the supplied "buffer" is smaller than the requested resource.
- **Tool_Err.Tool_Err**  Raised if an internal error is trapped.

---

**Tool_Res.Rfread example**

```sql
/*
** This examples uses Tool_Res.Rfread
*/
PROCEDURE ban_res IS
  resfileh   TOOL_RES.RFHANDLE;
  res1       VARCHAR2(20);
  ...
END;
```

---

150  Forms Developer and Reports Developer Built-in Package Reference
BEGIN
/*  Open a resource file */
resfileh := TOOL_RES.RFOPEN
('C:\resource\test.res');
/*  Read resource string 'banner' from file */
res1 := TOOL_RES.RFREAD(resfileh,'banner');
...

TEXT_IO.PUT_LINE(res1);
...
END;
Index

B

built-in packages
about Oracle Developer 1
building resource files 14
DDE 3, 4, 5, 6, 7, 29
Debug 8, 39
EXEC_SQL 16, 17, 18, 19, 45, 77
List 8, 79
OLE2 8, 85
Ora_Ffi 8, 99
Ora_Nls 9, 11, 117
Ora_Prof 12, 125
Text_IO 12, 13, 131
Tool_Env 13, 139
Tool_Err 13, 14, 141
Tool_Res 14, 147
built-in packages: 1, 3, 8, 9, 11, 12, 13, 14, 16, 29, 39, 45, 77, 79, 85, 99, 117, 125, 131, 139, 141, 147

C

connection handles 17
cursor handles 17

D
database connection
when changing primary 77
database connection: 77
DDE package
about 3
exceptions 6
predefined data formats 4
procedures and functions 29
DDE package: 3, 4, 6, 29
Debug package
about 8
procedures and functions 39
Debug package: 8, 39

E

exceptions
DDE predefined 6
Debug.Break 39
EXEC_SQL predefined 17
List.Fail 81
OLE2.OLE_Error 96
OLE2.OLE_Not_Supported 96
Ora_Ffi.Ffi_Error 99
Ora_Nls.Bad_Attribute 118
Ora_Nls.No_Item 122
Ora_Nls.Not_Found 122
Ora_Prof.Bad_Timer 125
Tool_Err.Tool_Error 144
Tool_Res.Bad_File_Handle 147
Tool_Res.Buffer_Overflow 148
Tool_Res.File_Not_Found 149
Tool_Res.No_Resource 149
exceptions: 6, 17, 39, 81, 96, 99, 118, 122, 125, 144, 147

EXEC_SQL package
about 16
examples using constructs 19
exceptions 17
procedures and functions 45
retrieving multiple result sets 17
EXEC_SQL package: 16, 17, 19, 45, 77

L

List package
about 8
procedures and functions 79
List package: 8, 79
O

OLE2 package
    about 8
    procedures and functions 85
OLE2 package: 8, 85

Ora_Ffi package
    about 8
    procedures and functions 99
Ora_Ffi package: 8, 99

Ora_Nls package
    about 9
    character constants 9
    numeric constants 11
    procedures and functions 117
Ora_Nls package: 9, 11, 117

Ora_Prof package
    about 12
    procedures and functions 125
Ora_Prof package: 12, 125

P

primary database connection
    when changing 77

R

resource files
    building 14
resource files: 14
RESPA21 utility 15
RESPR21 utility 14

T

Text_IO package
    about 12
    example using constructs 13
    procedures and functions 131
Text_IO package: 12, 13, 131

Tool_Env package
    about 13
    example using constructs 13
    procedures and functions 141
Tool_Env package: 13, 141

Tool_Err package
    about 13
    building resource files 14
    procedures and functions 147
Tool_Err package: 13, 141

Tool_Res package
    about 14
    building resource files 14
    procedures and functions 147
Tool_Res package: 14, 147

Tool_Err package
    about 13
    example using constructs 13
    procedures and functions 141
Tool_Err package: 13, 141

Tool_Res package
    about 14
    building resource files 14
    procedures and functions 147
Tool_Res package: 14, 147

Forms Developer and Reports Developer Built-in Package Reference