

Oracle9i

Globalization Support Guide

Release 1 (9.0.1)

June 2001

Part No. A90236-01

ORACLE®

Oracle 9i Globalization Support Guide, Release 1 (9.0.1)

Part No. A90236-01

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

Primary Author: Paul Lane

Contributors: Cathy Baird, Winson Chu, Jessica Fan, Yu Gong, Claire Ho, Simon Law, Peter Linsley, Den Raphaely, Shige Takeda, Linus Tanaka, Makoto Tozawa, Barry Trute, Michael Yau, Hiro Yoshioka, Sergiusz Wolicki, Simon Wong

Graphic Designer: Valarie Moore

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, 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.

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs 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. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs 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 the Programs are delivered to the U.S. Government or anyone licensing or using the Programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs including documentation, 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 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle is a registered trademark, and Oracle9i, Enterprise Manager, Pro*COBOL, SQL*Forms, SQL*Plus, Oracle Call Interface, Oracle Forms, Oracle Net Services, PL/SQL, Pro*C, Pro*C/C++, and Trusted Oracle are registered trademarks or trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

Contents

Send Us Your Comments	xv
Preface.....	xvii
1 Globalization Support	
Oracle Server Globalization Support Architecture.....	1-2
Locale-Independent Operation.....	1-2
Multitier Architecture	1-4
Unicode	1-5
Globalization Support Features.....	1-5
Language Support	1-6
Territory Support.....	1-6
Date and Time Formats	1-6
Monetary and Numeric Formats.....	1-7
Calendars	1-7
Linguistic Sorting.....	1-7
Character Set Support	1-7
Customization	1-8
2 Choosing a Character Set	
Character Set Encoding.....	2-2
What is an Encoded Character Set?	2-2
Which Characters to Encode?	2-3
How Many Languages Does a Character Set Support?	2-4

How are Characters Encoded?	2-8
Oracle's Naming Convention for Character Sets	2-10
Choosing an Oracle Database Character Set	2-10
Interoperability with System Resources and Applications	2-11
Character Set Conversion	2-11
Database Schemas.....	2-12
Performance Implications.....	2-12
Restriction	2-12
Choosing an Oracle NCHAR Character Set.....	2-13
Restrictions on Character Sets Used to Express Names and Text	2-13
Summary of Datatypes and Supported Encoding Schemes.....	2-15
Changing the Character Set After Database Creation.....	2-15
Monolingual Database Scenario.....	2-16
Character Set Conversion	2-16
Multilingual Database Scenarios.....	2-18
Restricted Multilingual Support.....	2-18
Unrestricted Multilingual Support.....	2-19

3 Setting Up a Globalization Support Environment

Setting NLS Parameters	3-2
Choosing a Locale with the NLS_LANG Initialization Parameter	3-4
Specifying NLS_LANG as an Environment Variable.....	3-6
NLS_LANG Examples	3-6
Overriding Language and Territory Specifications	3-6
NLS Database Parameters	3-7
Checking NLS Parameters.....	3-7
NLS Views	3-7
OCI Functions.....	3-8
Language and Territory Parameters.....	3-8
Date and Time Parameters	3-14
Date Formats	3-14
Time Formats.....	3-17
Calendar Parameter	3-21
Calendar Formats.....	3-21
NLS_CALENDAR	3-23

Numeric Parameters	3-24
Numeric Formats	3-24
NLS_NUMERIC_CHARACTERS	3-25
Monetary Parameters	3-26
Currency Formats	3-26
NLS_CURRENCY	3-26
NLS_ISO_CURRENCY	3-28
NLS_DUAL_CURRENCY	3-29
NLS_MONETARY_CHARACTERS	3-30
NLS_CREDIT	3-31
NLS_DEBIT	3-31
Linguistic Sorting Parameters	3-31
NLS_SORT	3-32
NLS_COMP	3-33
NLS_LIST_SEPARATOR	3-34
Character Set Parameter	3-34
NLS_NCHAR_CONV_EXCP	3-34

4 Linguistic Sorting

Overview of Oracle's Sorting Capabilities	4-2
Using Binary Sorts	4-2
Using Linguistic Sorts	4-2
Monolingual Linguistic Sorts	4-3
Multilingual Linguistic Sorts	4-3
General Linguistic Sorting Information	4-6
Using Linguistic Indexes	4-9
Linguistic Indexes for Multiple Languages	4-10
Requirements for Linguistic Indexes	4-11
Case-Insensitive Searching	4-12
Customizing Linguistic Sorts	4-12

5 Supporting Multilingual Databases with Unicode

Overview of Unicode	5-2
What is Unicode?	5-2
Unicode Encoding	5-2

Implementing a Unicode Solution in the Database	5-4
Enabling Multilingual Support with Unicode Databases.....	5-4
Enabling Multilingual Support with Unicode Datatypes.....	5-5
How to Choose Between a Unicode Database and a Unicode Datatype Solution.....	5-7
Comparison of Unicode Solutions	5-9
Unicode Case Studies	5-12
Migrating Data to Unicode	5-14
Migrating to a Unicode Database.....	5-15
Migrating to Use the NCHAR Datatypes.....	5-16
Designing Database Schemas to Support Multiple Languages	5-17
Specifying Column Limits.....	5-17
Storing Data of Multiple Languages.....	5-18
Storing Documents in LOBs.....	5-19

6 Unicode Programming

Overview of Unicode Programming	6-2
Database Access Product Stack and Unicode.....	6-2
SQL and PL/SQL Programming with Unicode	6-4
Using the UTL_FILE Package with NCHAR.....	6-10
OCI Programming with Unicode	6-11
OCI Unicode Code Conversion	6-11
When NLS_LANG is Set to UTF8 or AL32UTF8 in OCI.....	6-15
Binding and Defining SQL CHAR Datatypes in OCI	6-15
Binding and Defining SQL NCHAR Datatypes in OCI	6-16
Binding and Defining CLOB and NCLOB Unicode Data in OCI.....	6-17
Unicode Mode in OCI	6-18
Pro*C/C++ Programming with Unicode	6-20
Pro*C/C++ Data Conversion in Unicode	6-20
Using the VARCHAR Datatype	6-21
Using the NVARCHAR Datatype	6-22
Using the UVARCHAR Datatype	6-22
JDBC and SQLJ Programming with Unicode	6-23
Java String Bind and Define in Unicode.....	6-23
JDBC Restriction.....	6-25
Java Data Conversion in Unicode	6-25

ODBC and OLEDB Programming with Unicode	6-27
Unicode-Enabled Drivers in ODBC and OLEDB.....	6-27
OCI Dependency in Unicode	6-27
ODBC and OLEDB Code Conversion in Unicode	6-28
ODBC Unicode Datatypes.....	6-29
OLEDB Unicode Datatypes.....	6-30
ADO Access.....	6-31

7 SQL Programming

Locale-Dependent SQL Functions.....	7-2
Default Specifications.....	7-3
Specifying Parameters.....	7-3
Unacceptable Parameters	7-5
CONVERT Function.....	7-5
INSTR, LENGTH, and SUBSTR Functions and Character Sets	7-6
LIKE Conditions and Character Sets	7-8
Character Set SQL Functions	7-9
NLSSORT Function	7-9
Time/Date/Calendar Formats.....	7-12
Date Formats	7-12
Numeric Formats.....	7-13
Miscellaneous Topics	7-14
The Concatenation Operator.....	7-14

8 OCI Programming

Using the OCI NLS Functions.....	8-2
NLS Language Information Retrieval	8-2
OCI_Nls_GetInfo	8-3
OCI_Nls_MaxBufSz	8-7
NLS Language Information Retrieval Sample Code.....	8-7
String Manipulation in OCI.....	8-8
OCI_MultiByteToWideChar	8-10
OCI_MultiByteInSizeToWideChar	8-11
OCI_WideCharToMultiByte	8-12
OCI_WideCharInSizeToMultiByte	8-12

OCIWideCharToLower.....	8-13
OCIWideCharToUpper.....	8-14
OCIWideCharStrcmp.....	8-14
OCIWideCharStrncmp.....	8-15
OCIWideCharStrcat.....	8-16
OCIWideCharStrchr.....	8-17
OCIWideCharStrcpy.....	8-17
OCIWideCharStrlen.....	8-18
OCIWideCharStrncat.....	8-18
OCIWideCharStrncpy.....	8-19
OCIWideCharStrrchr.....	8-19
OCIWideCharStrCaseConversion.....	8-20
OCIWideCharDisplayLength.....	8-21
OCIWideCharMultiByteLength.....	8-21
OCIMultiByteStrcmp.....	8-22
OCIMultiByteStrncmp.....	8-22
OCIMultiByteStrcat.....	8-23
OCIMultiByteStrcpy.....	8-24
OCIMultiByteStrlen.....	8-24
OCIMultiByteStrncat.....	8-25
OCIMultiByteStrncpy.....	8-25
OCIMultiByteStrnDisplayLength.....	8-26
OCIMultiByteStrCaseConversion.....	8-27
String Manipulation Sample Code.....	8-27
Character Classification in OCI.....	8-28
OCIWideCharIsAlnum.....	8-29
OCIWideCharIsAlpha.....	8-29
OCIWideCharIsCntrl.....	8-30
OCIWideCharIsDigit.....	8-30
OCIWideCharIsGraph.....	8-31
OCIWideCharIsLower.....	8-31
OCIWideCharIsPrint.....	8-32
OCIWideCharIsPunct.....	8-32
OCIWideCharIsSpace.....	8-32
OCIWideCharIsUpper.....	8-33

OCIWideCharIsXdigit	8-33
OCIWideCharIsSingleByte.....	8-34
Character Set Conversion in OCI	8-35
OCICharSetToUnicode	8-36
OCIUnicodeToCharSet	8-36
OCICharSetConversionIsReplacementUsed	8-37
Messaging Mechanism in OCI.....	8-39
OCIMessageOpen.....	8-39
OCIMessageGet	8-40
OCIMessageClose.....	8-41
LMSGEN	8-42
Text Message File Format.....	8-42
Message Example	8-43

9 Java Programming

Overview of Oracle9i Java Support.....	9-2
JDBC.....	9-3
Accessing SQL CHAR Datatypes Using JDBC.....	9-4
Accessing SQL NCHAR Datatypes Using JDBC	9-7
Using the oracle.sql.CHAR Class	9-8
Retrieving Data to oracle.sql.CHAR Class.....	9-8
NLS Restrictions	9-10
SQLJ.....	9-13
Using Unicode characters in SQLJ programs.....	9-13
Using the oracle.sql.NString class.....	9-14
Java Virtual Machine.....	9-14
Java Stored Procedures	9-16
Java Servlets and Java Server Pages	9-18
CORBA and EJB.....	9-21
CORBA ORB.....	9-21
Enterprise Java Beans.....	9-25
Configurations for Multilingual Applications.....	9-28
Configuring a Multilingual Database.....	9-28
Globalizing the Java Server Objects	9-29
Clients of Different Languages.....	9-30

Multilingual Demo Applications in SQLJ.....	9-31
The Database Schema.....	9-31
Java Stored Procedures	9-32
The SQLJ Client.....	9-35

10 Character Set Scanner Utility

Overview of Choosing and Migrating Character Sets.....	10-2
Data Truncation	10-2
Character Set Conversions	10-4
Database Character Set Migration.....	10-6
Data Scanning.....	10-7
Conversion of Data.....	10-7
What is the Character Set Scanner Utility?	10-9
Conversion Tests on Character Data.....	10-10
Access Privileges.....	10-10
Restrictions.....	10-10
Database Containing Data From Two or More Character Sets.....	10-11
Database Containing Data Not From the Database Character Set	10-11
Scan Modes in the Scanner	10-11
Full Database Scan.....	10-12
User Tables Scan	10-12
Single Table Scan	10-12
Using The Scanner.....	10-12
Before Using the Scanner.....	10-13
Compatibility.....	10-13
Invoking the Scanner.....	10-14
Getting Online Help for the Scanner	10-14
The Parameter File	10-15
Scanner Parameters.....	10-16
ARRAY	10-16
BOUNDARIES.....	10-17
CAPTURE	10-17
FEEDBACK.....	10-17
FROMCHAR.....	10-18
FROMNCHAR	10-18

FULL	10-18
HELP	10-19
LASTRPT	10-19
LOG	10-19
MAXBLOCKS.....	10-20
PARFILE	10-20
PROCESS	10-21
SUPPRESS.....	10-21
TABLE	10-21
TOCHAR.....	10-22
TONCHAR	10-22
USER.....	10-22
USERID	10-22
Sample Scanner Sessions	10-23
Sample Session of Full Database Scan.....	10-23
Sample Session of User Tables Scan	10-24
Sample Session of Single Table Scan.....	10-25
Scanner Reports	10-26
Database Scan Summary Report	10-27
Individual Exception Report.....	10-33
Storage and Performance Considerations in the Scanner.....	10-35
Storage Considerations	10-35
Performance Considerations.....	10-36
Scanner Utility Reference Material.....	10-37
Scanner Views	10-37
Scanner Messages	10-40

11 Oracle Locale Builder Utility

Overview of the Locale Builder Utility	11-2
Configuring Unicode Fonts for the Locale Builder	11-2
The Locale Builder Interface	11-3
Locale Builder General Screens	11-4
Restrictions	11-5
Setting the Language Definition with the Locale Builder.....	11-8
Setting the Territory Definition with the Locale Builder.....	11-11

Setting the Character Set Definition with the Locale Builder.....	11-16
Character Sets with User-Defined Characters	11-17
Oracle's Character Set Conversion Architecture	11-18
Unicode 3.1 Private Use Area	11-19
UDC Cross References	11-19
Character Set Definition File Conventions.....	11-19
Locale Builder Character Set Scenario	11-20
Sorting with the Locale Builder	11-25
Changing the Sort Order for Accented Characters.....	11-28
Changing the Sort Order for One Accented Character	11-31
Generating NLB Files	11-33
Using the New NLB Files	11-34

12 Customizing Locale Data

Customizing Character Sets	12-2
Character Set Customization Example	12-2
Using User-Defined Character Sets and Java	12-4
Customizing Time Zone Data.....	12-6
Customizing Calendars.....	12-7
NLS Calendar Utility.....	12-7
NLS Data Installation Utility.....	12-8
Syntax	12-8
Return Codes	12-9
Usage	12-9

A Locale Data

Languages	A-2
Translated Messages.....	A-4
Territories.....	A-5
Character Sets	A-6
Asian Language Character Sets	A-8
European Language Character Sets	A-9
Middle Eastern Language Character Sets	A-15
Universal Character Sets.....	A-18
Character Set Conversion Support.....	A-18

Subsets and Supersets A-19

Linguistic Sorting A-22

Calendar Systems..... A-26

Obsolete Locale Data A-28

AL24UTFSS Character Set Desupport..... A-30

B Unicode Character Code Assignments

Unicode Character Code Assignments B-2

UTF-16 Encoding B-3

UTF-8 Encoding B-3

Glossary

Index

Send Us Your Comments

Oracle9i Globalization Support Guide, Release 1 (9.0.1)

Part No. A90236-01

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

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

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

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
Oracle Corporation
Server Technologies Documentation
500 Oracle Parkway, Mailstop 40p11
Redwood Shores, CA 94065
USA

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

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

Preface

This manual provides information about Oracle's Globalization Support capabilities.

This preface contains these topics:

- [Audience](#)
- [Organization](#)
- [Related Documentation](#)
- [Conventions](#)

Audience

This manual is written for database administrators, system administrators, and database application developers who need to ensure that their database or applications include globalization support.

To use this document, you need to be familiar with relational database concepts, basic Oracle server concepts, and the operating system environment under which you are running Oracle.

In addition to administrators, experienced users of Oracle and advanced database application designers will find information in this manual useful. However, database application developers should also refer to the *Oracle9i Application Developer's Guide - Fundamentals* and to the documentation for the tool or language product they are using to develop Oracle database applications.

Organization

This document contains:

Chapter 1, "Globalization Support"

This chapter contains an overview of globalization and Oracle's approach to globalization.

Chapter 2, "Choosing a Character Set"

This chapter describes how to choose a character set.

Chapter 3, "Setting Up a Globalization Support Environment"

This chapter contains sample scenarios for enabling globalization capabilities.

Chapter 4, "Linguistic Sorting"

This chapter describes linguistic sorting.

Chapter 5, "Supporting Multilingual Databases with Unicode"

This chapter describes Unicode considerations for databases.

Chapter 6, "Unicode Programming"

This chapter describes how to program in a Unicode environment.

Chapter 7, "SQL Programming"

This chapter describes globalization considerations for SQL programming.

Chapter 8, "OCI Programming"

This chapter describes globalization considerations for OCI programming.

Chapter 9, "Java Programming"

This chapter describes globalization considerations for Java.

Chapter 10, "Character Set Scanner Utility"

This chapter describes how to use the Character Set Scanner utility to analyze character data.

Chapter 11, "Oracle Locale Builder Utility"

This chapter explains how to use the Oracle Locale Builder utility to customize locales.

Chapter 12, "Customizing Locale Data"

This chapter shows how to customize NLS data objects.

Appendix A, "Locale Data"

This chapter describes the languages, territories, character sets, and other locale data supported by the Oracle server.

Appendix B, "Unicode Character Code Assignments"

This chapter lists Unicode code point values.

Glossary

The glossary contains definitions of globalization support terms.

Related Documentation

For more information, see this Oracle resource:

- *Oracle9i Application Developer's Guide - Fundamentals*

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

<http://oraclestore.oracle.com/>

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

<http://www.oraclebookshop.com/>

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

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

<http://technet.oracle.com/membership/index.htm>

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

<http://technet.oracle.com/docs/index.htm>

Conventions

This section describes the conventions used in the text and code examples of the this documentation set. It describes:

- [Conventions in Text](#)
- [Conventions in Code Examples](#)

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	The C datatypes such as ub4 , sword , or OCINumber are valid. When you specify this clause, you create an index-organized table .
<i>Italics</i>	Italic typeface indicates book titles, emphasis, syntax clauses, or placeholders.	<i>Oracle9i Database Concepts</i> You can specify the <i>parallel_clause</i> . Run <i>Uold_release</i> .SQL where <i>old_release</i> refers to the release you installed prior to upgrading.
UPPERCASE monospace (fixed-width font)	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, user names, and roles.	You can specify this clause only for a NUMBER column. You can back up the database using the BACKUP command. Query the TABLE_NAME column in the USER_TABLES table in the data dictionary view. Specify the ROLLBACK_SEGMENTS parameter. Use the DBMS_STATS.GENERATE_STATS procedure.
lowercase monospace (fixed-width font)	Lowercase monospace typeface indicates executables and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, user names and roles, program units, and parameter values.	Enter sqlplus to open SQL*Plus. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to true. Connect as oe user.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[]	Brackets enclose one or more optional items. Do not enter the brackets.	DECIMAL (digits [, precision])
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	{ENABLE DISABLE}
	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	{ENABLE DISABLE} [COMPRESS NOCOMPRESS]
...	Horizontal ellipsis points indicate either: <ul style="list-style-type: none"> ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code 	CREATE TABLE ... AS subquery; SELECT col1, col2, ... , coln FROM employees;
.	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as it is shown.	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
<i>Italics</i>	Italicized text indicates variables for which you must supply particular values.	CONNECT SYSTEM/ <i>system_password</i>
UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.	SELECT last_name, employee_id FROM employees; sqlplus hr/hr

Documentation Accessibility

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

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

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

Globalization Support

This chapter provides an overview of Oracle Globalization Support. It includes the following topics:

- [Oracle Server Globalization Support Architecture](#)
- [Globalization Support Features](#)

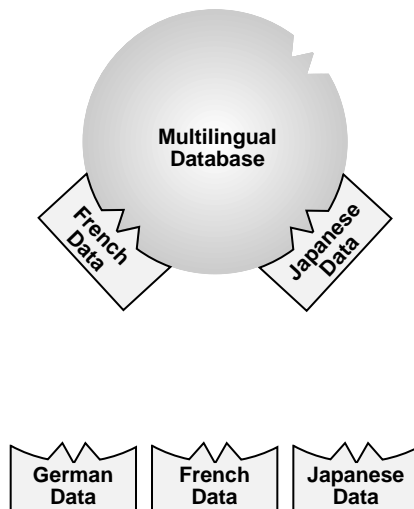
Oracle Server Globalization Support Architecture

Oracle's Globalization Support architecture allows you to store, process, and retrieve data in native languages. It ensures that database utilities, error messages, sort order, date, time, monetary, numeric, and calendar conventions automatically adapt to any native language and locale. In the past, Oracle's Globalization Support capabilities were referred to as National Language Support (NLS) features. National Language Support is a subset of Globalization Support because the Oracle database can do more than handle one national language or store data in one character set. Globalization Support allows you to develop multilingual applications and software products that can be accessed and run from anywhere in the world simultaneously without modification. The applications can render content in native users' languages and locale preferences.

Locale-Independent Operation

Oracle's Globalization Support architecture is implemented with the Oracle NLS Runtime Library (NLSRTL). The NLS Runtime library provides a comprehensive suite of language-independent functions that allow proper text and character processing and language convention manipulations. Behavior of these functions for a specific language and territory is governed by a set of locale-specific data identified and loaded at runtime.

[Figure 1-1](#) illustrates loading locale-specific data at runtime. For example, French and Japanese locale data are loaded.

Figure 1–1 Loading Locale-specific Data at Runtime

The locale-specific data is stored in a directory specified by the `ORA_NLS*` environment variable. For each new release of the Oracle database, there is a different `ORA_NLS` data directory. For Oracle9i, the `ORA_NLS33` directory is used. For example, on most UNIX platforms, the `ORA_NLS33` environment variable should be set to `$ORACLE_HOME/ocommon/nls/admin/data`. On Win32 platforms, use the default that is set by the installer if the `ORACLE_HOME` directory contains just one release of Oracle.

Table 1–1 Location of NLS Data

Release	Environment Variable
7.2	<code>ORA_NLS</code>
7.3	<code>ORA_NLS32</code>
8.0, 8.1, 9.0.1	<code>ORA_NLS33</code>

If your system is running in a multi-version Oracle environment, you must ensure that the appropriate `ORA_NLS*` variable (for example, `ORA_NLS33`) is set and that the corresponding NLS datafiles for that release are available.

A boot file is used to determine the availability of the NLS objects that can be loaded. Oracle supports both system and user boot files. The user boot file gives

you the flexibility to tailor what NLS locale objects will be available for the database, thus helping you control memory consumption. Also, new locale data can be added and some locale data components can be customized.

See Also: [Chapter 11, "Oracle Locale Builder Utility"](#) for more information about data customization

Multitier Architecture

The Oracle9i database is implemented using a multitier architecture. The language-dependent operations are controlled by several parameters and environment variables on both the client and the database server. On the database server, each session started on behalf of a client may run in the same or a different locale, and have the same or different language requirements specified.

A database itself also has a set of session-independent NLS parameters specified at its creation time. Two of the parameters specify the database character set and the national (Unicode) character set. The parameters specify the character set used to store text data in the database. Other parameters, like language and territory, are used to evaluate `CHECK` constraints.

If the client session and the database server specify different character sets, the Oracle9i database converts character set strings automatically.

From a Globalization Support perspective, all applications, even those running on the same physical machine as the Oracle instance, are considered clients. For example, when SQL*Plus is started by the Unix user who owns the Oracle software from the Oracle home in which the RDBMS software is installed, and SQL*Plus connects to the database through an adapter by specifying the `ORACLE_SID`, SQL*Plus is considered a client and its behavior is ruled by client-side NLS parameters. Another example is when the middle tier is an application server and the different sessions spawned by it are considered to be separate client sessions.

When a client application is started, it initializes its client NLS environment from environment settings. All NLS operations performed locally are executed using these settings. Examples of local NLS operations are display formatting (using item format masks) in Oracle Developer applications or user OCI code executing NLS OCI functions with OCI environment handles.

See Also: [Chapter 8, "OCI Programming"](#)

When the application connects to a database, a session is created on the server. The new session initializes its NLS environment from NLS instance parameters specified in the initialization parameter file. These settings can be subsequently changed by

an `ALTER SESSION` statement. The statement changes only the session NLS environment. It does not change the local client NLS environment. The session NLS settings are used to process SQL and PL/SQL statements that are executed on the server.

Immediately after the connection, if the `NLS_LANG` environment setting is defined on the client side, an implicit `ALTER SESSION` statement synchronizes the client and the session NLS environments.

See Also: [Chapter 3, "Setting Up a Globalization Support Environment"](#)

Unicode

Unicode is a universal encoded character set that allows you to store information from any language using a single character set. Unicode provides a unique code value for every character, regardless of the platform, program, or language.

Incorporating Unicode into client-server or multitiered applications and websites offers significant cost savings over the use of legacy character sets. Unicode enables a single software product or a single website to be targeted across multiple platforms, languages, and countries without re-engineering. It also allows data to be transported through many different systems without corruption.

See Also:

- [Chapter 5, "Supporting Multilingual Databases with Unicode"](#)
- [Chapter 6, "Unicode Programming"](#)

Globalization Support Features

Oracle's standard features include

- [Language Support](#)
- [Territory Support](#)
- [Date and Time Formats](#)
- [Monetary and Numeric Formats](#)
- [Calendars](#)
- [Linguistic Sorting](#)
- [Character Set Support](#)

- [Customization](#)

Language Support

The Oracle9i database allows you to store, process, and retrieve data in native languages. The languages that can be stored in an Oracle9i database are all languages written in scripts that are encoded by Oracle-supported character sets. Through the use of Unicode databases and datatypes, Oracle9i supports most contemporary languages.

Additional support is available for a subset of the languages. The Oracle9i database knows, for example, how to display dates using translated month names or how to sort text data according to cultural conventions.

When this manual uses the term **language support**, it refers to the additional language-dependent functionality, not to the ability to store text of the given language.

For some of the supported languages, Oracle provides translated error messages and a translated user interface of the database utilities.

See Also:

- ["Languages"](#) on page A-2 for a complete list of Oracle language names and abbreviations
- ["Translated Messages"](#) on page A-4

Territory Support

The Oracle9i database supports cultural conventions that are specific to geographical locations. The default local time format, date format, numeric and monetary conventions depend on the local territory setting. By setting different NLS parameters, the database session can use different cultural settings. For example, you can set British pound sterling (GBP) as the primary currency and the Japanese yen (JPY) as the secondary currency for a given database session even when the territory is defined as AMERICA.

See Also: ["Territories"](#) on page A-5

Date and Time Formats

Different conventions for displaying the hour, day, month, and year can be handled in local formats. For example, in the United Kingdom, the date is displayed using the DD-MON-YYYY format, while Japan commonly uses the YYYY-MON-DD format.

Time zones and daylight saving support are also available.

See Also:

- [Chapter 3, "Setting Up a Globalization Support Environment"](#)
- *Oracle9i SQL Reference*
- *Oracle9i Database Administrator's Guide*

Monetary and Numeric Formats

Currency, credit, and debit symbols can be represented in local formats. Radix symbols and thousands separators can be defined by locales. For example, in the US, the decimal point is a dot (.), while it is a comma (,) in France. Therefore, the amount \$1,234 has different meanings in different countries.

Calendars

Many different calendar systems are in use around the world. Oracle supports seven different calendar systems: Gregorian, Japanese Imperial, ROC Official (Republic of China), Thai Buddha, Persian, English Hijrah, and Arabic Hijrah.

See Also: ["Calendar Systems"](#) on page A-26 for a complete list of calendars

Linguistic Sorting

Oracle9i provides linguistic definitions for culturally accurate sorting and case conversion. Some of the definitions have two versions. The basic definition treats strings as sequences of independent characters. The extended definition recognizes pairs of characters that should be treated as special cases.

Strings that are converted to upper case or lower case using the basic definition always retain their lengths. Strings converted using the extended definition may get longer or shorter.

See Also: [Chapter 4, "Linguistic Sorting"](#)

Character Set Support

Oracle supports a large number of single-byte, multibyte, and fixed-width encoding schemes that are based on national, international, and vendor-specific standards.

See Also: ["Character Sets"](#) on page A-6 for a complete list of supported character sets

Customization

You can customize locale data and calendar settings.

Locale Data

Oracle allows you to customize all locale data such as language, character set, territory, or linguistic sort using the Oracle Locale Builder.

See Also: [Chapter 11, "Oracle Locale Builder Utility"](#)

Calendar

You can define ruler eras for imperial calendars and deviation days for lunar calendars.

See Also: ["Customizing Calendars"](#) on page 12-7

Choosing a Character Set

This chapter explains how to choose a character set. It includes the following topics:

- [Character Set Encoding](#)
- [Choosing an Oracle Database Character Set](#)
- [Monolingual Database Scenario](#)
- [Multilingual Database Scenarios](#)

Character Set Encoding

When computer systems process characters, they use numeric codes instead of the graphical representation of the character. For example, when the database stores the letter A, it actually stores a numeric code that is interpreted by software as that letter. These numeric codes are important in all databases. They are especially important when working in a global environment because of the need to convert between different character sets.

What is an Encoded Character Set?

An encoded character set is specified when you create a database. The choice of character set determines what languages can be represented in the database. This choice influences how you create the database schema and develop applications that process character data. It also influences interoperability with operating system resources and database performance.

A group of characters (for example, alphabetic characters, ideographs, symbols, punctuation marks, and control characters) can be encoded as an encoded character set. An encoded character set assigns unique numeric codes to each character in the character repertoire. [Table 2–1](#) shows examples of characters that are assigned a numeric code value.

Table 2–1 Encoded Characters in the ASCII Character Set

Character	Description	Code Value
!	Exclamation Mark	21
#	Number Sign	23
\$	Dollar Sign	24
1	Number 1	31
2	Number 2	32
3	Number 3	33
A	Uppercase A	41
B	Uppercase B	42
C	Uppercase C	43
a	Lowercase a	61
b	Lowercase b	62
c	Lowercase c	63

There are many different coded character sets used throughout the computer industry. Oracle supports most national, international, and vendor-specific encoded character set standards. The complete list of character sets supported by Oracle is listed in [Appendix A, "Locale Data"](#). Character sets differ in the following ways:

- The number of characters available
- The characters available (the character repertoire)
- The writing scripts and the languages represented
- The code values assigned to each character
- The encoding scheme used to represent a character

These differences are discussed throughout this chapter.

Which Characters to Encode?

When you choose a character set, first decide what languages you wish to store in the database. The characters that are encoded in a character set depend on the writing systems that are represented.

Writing Systems

A writing system can be used to represent a language or group of languages. For the purposes of this book, writing systems can be classified into two categories: phonetic and ideographic.

Phonetic Writing Systems Phonetic writing systems consist of symbols that represent different sounds associated with a language. Greek, Latin, Cyrillic, and Devanagari are all examples of phonetic writing systems based on alphabets. Note that alphabets can represent more than one language. For example, the Latin alphabet can represent many Western European languages such as French, German, and English.

Characters associated with a phonetic writing system (alphabet) can typically be encoded in one byte because the character repertoire is usually smaller than 256 characters.

Ideographic Writing Systems Ideographic writing systems consist of ideographs or pictographs that represent the meaning of a word, not the sounds of a language. Chinese and Japanese are examples of ideographic writing systems that are based on tens of thousands of ideographs. Languages that use ideographic writing systems may use a **syllabary** as well. Syllabaries provide a mechanism for

communicating phonetic information along with the pictographs when necessary. For instance, Japanese has two syllabaries: Hiragana, normally used for grammatical elements, and Katakana, normally used for foreign and onomatopoeic words.

Characters associated with an ideographic writing system typically must be encoded in more than one byte because the character repertoire has tens of thousands of characters.

Punctuation, Control Characters, Numbers, and Symbols In addition to encoding the script of a language, other special characters, such as punctuation marks, need to be encoded such as punctuation marks (for example, commas, periods, and apostrophes), numbers (for example, Arabic digits 0-9), special symbols (for example, currency symbols and math operators) and control characters for computers (for example, carriage returns, tabs, and `NULL`).

Writing Direction Most Western languages are written left to right from the top to the bottom of the page. East Asian languages are usually written top to bottom from the right to the left of the page, though exceptions are frequently made for technical books translated from Western languages. Arabic and Hebrew are written right to left from the top to the bottom.

Another consideration is that numbers reverse direction in Arabic and Hebrew. So even though the text is written right to left, numbers within the sentence are written left to right. For example, "I wrote 32 books" would be written as "skoob 32 etorw I". Regardless of the writing direction, Oracle stores the data in logical order. Logical order means the order used by someone typing a language, not how it looks on the screen.

How Many Languages Does a Character Set Support?

Different character sets support different character repertoires. Because character sets are typically based on a particular writing script, they can thus support different languages. When character sets were first developed in the United States, they had a limited character repertoire and even now there can be problems using certain characters across platforms. The following `CHAR` and `VARCHAR` characters are represented in all Oracle database character sets and transportable to any platform:

- Uppercase and lowercase English characters A-Z and a-z
- Arabic digits 0-9
- The following punctuation marks:

%	‘	’	(
)	*	+	-
,	.	/	\
:	;	<	>
=	!	_	&
~	{	}	
^	?	\$	#
@	"	[]

- The following control characters:

- <space>
- <horizontal tab>
- <vertical tab>
- <form feed>

If you are using:

- Characters outside this set
- Unicode datatypes (NCHAR or NVARCHAR characters)

then take care that your data is in well-formed strings.

During conversion from one character set to another, Oracle expects CHAR and VARCHAR items to be well-formed strings encoded in the declared database character set. If you put other values into the string (for example, using the CHR or CONVERT function), the values may be corrupted when they are sent to a database with a different character set.

If you are currently using only two or three well-established character sets, you may not have experienced any problems with character conversion. However, as your enterprise grows and becomes more global, problems may arise with such conversions. Therefore, Oracle Corporation recommends that you use Unicode databases and datatypes.

See Also: [Chapter 5, "Supporting Multilingual Databases with Unicode"](#)

ASCII Encoding

The ASCII and IBM EBCDIC character sets support a similar character repertoire, but assign different code values to some of the characters. [Table 2–2](#) shows how ASCII is encoded. Row and column headings denote hexadecimal digits. To find the encoded value of a character, read the column number followed by the row number. For example, the value of the character A is 0x41.

Table 2–2 7-Bit ASCII Coded Character Set

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	TAB	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Over the years, character sets evolved to support more than just monolingual English in order to meet the growing needs of users around the world. New character sets were quickly created to support other languages. Typically, these new character sets supported a group of related languages, based on the same script. For example, the ISO 8859 character set series was created to support different European languages.

Table 2–3 ISO 8859 Character Sets

Standard	Languages Supported
ISO 8859-1	Western European (Albanian, Basque, Breton, Catalan, Danish, Dutch, English, Faeroese, Finnish, French, German, Greenlandic, Icelandic, Irish Gaelic, Italian, Latin, Luxembourgish, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, Swedish)
ISO 8859-2	Eastern European (Albanian, Croatian, Czech, English, German, Hungarian, Latin, Polish, Romanian, Slovak, Slovenian, Serbian)
ISO 8859-3	Southeastern European (Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, Turkish)
ISO 8859-4	Northern European (Danish, English, Estonian, Finnish, German, Greenlandic, Latin, Latvian, Lithuanian, Norwegian, Sámi, Slovenian, Swedish)
ISO 8859-5	Eastern European (Cyrillic-based: Bulgarian, Byelorussian, Macedonian, Russian, Serbian, Ukrainian)
ISO 8859-6	Arabic
ISO 8859-7	Greek
ISO 8859-8	Hebrew
ISO 8859-9	Western European (Albanian, Basque, Breton, Catalan, Cornish, Danish, Dutch, English, Finnish, French, Frisian, Galician, German, Greenlandic, Irish Gaelic, Italian, Latin, Luxembourgish, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, Swedish, Turkish)
ISO 8859-10	Northern European (Danish, English, Estonian, Faeroese, Finnish, German, Greenlandic, Icelandic, Irish Gaelic, Latin, Lithuanian, Norwegian, Sámi, Slovenian, Swedish)
ISO 8859-13	Baltic Rim (English, Estonian, Finnish, Latin, Latvian, Norwegian)
ISO 8859-14	Celtic (Albanian, Basque, Breton, Catalan, Cornish, Danish, English, Galician, German, Greenlandic, Irish Gaelic, Italian, Latin, Luxembourgish, Manx Gaelic, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, Swedish, Welsh)
ISO 8859-15	Western European (Albanian, Basque, Breton, Catalan, Danish, Dutch, English, Estonian, Faroese, Finnish, French, Frisian, Galician, German, Greenlandic, Icelandic, Irish Gaelic, Italian, Latin, Luxembourgish, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, Swedish)

Character sets evolved and provided restricted multilingual support. They were restricted in the sense that they were limited to groups of languages based on similar scripts. More recently, there has been a push to remove boundaries and limitations on the character data that can be represented through the use of an unrestricted or universal character set. Unicode is one such universal character set that encompasses most major scripts of the modern world. The Unicode character

set provides support for a character repertoire of approximately 49,000 characters and continues to grow.

How are Characters Encoded?

Different types of encoding schemes have been created by the computer industry. The character set you choose affects what kind of encoding scheme will be used. This is important because different encoding schemes have different performance characteristics, and these characteristics can influence your database schema and application development requirements. The character set you choose will typically use one of the following types of encoding schemes:

- **Single-Byte Encoding Schemes**
 - 7-Bit Encoding Schemes
 - 8-Bit Encoding Schemes
- **Multibyte Encoding Schemes**
 - Fixed-Width Multibyte Encoding Schemes
 - Variable-Width Multibyte Encoding Schemes

Single-Byte Encoding Schemes

Single byte encoding schemes are the most efficient encoding schemes available. They take up the least amount of space to represent characters and are easy to process and program with because one character can be represented in one byte.

7-Bit Encoding Schemes Single-byte 7-bit encoding schemes can define up to 128 characters and normally support just one language. One of the most common single-byte character sets, used since the early days of computing, is ASCII (American Standard Code for Information Interchange).

8-Bit Encoding Schemes Single-byte 8-bit encoding schemes can define up to 256 characters and often support a group of related languages. One example is ISO 8859-1, which supports many Western European languages. [Figure 2-1](#) illustrates a typical 8-bit encoding scheme.

Figure 2–1 8-Bit Encoding Schemes

	0	1	2	3	4	5	6	7	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p	NBSP	°	À	Ð	à	ø
1	SOH	DC1	!	1	A	Q	a	q	¡	±	Á	Ñ	á	ñ
2	STX	DC2	"	2	B	R	b	r	¢	²	Â	Ò	â	ò
3	ETX	DC3	#	3	C	S	c	s	£	³	Ã	Ó	ã	ó
4	EOT	DC4	\$	4	D	T	d	t	¥	´	Ä	Ô	ä	ô
5	ENQ	NAK	%	5	E	U	e	u	¥	µ	Å	Ö	å	ö
6	ACK	SYN	&	6	F	V	f	v	¦	¶	Æ	Ø	æ	ø
7	BEL	ETB	'	7	G	W	g	w	§	·	Ç	×	ç	÷
8	BS	CAN	(8	H	X	h	x	"	¸	È	Ø	è	ø
9	HT	EM)	9	I	Y	i	y	@	¹	É	Ù	é	ù
A	NL	SUB	*	:	J	Z	j	z	ª	º	Ê	Ú	ê	û
B	VT	ESC	+	;	K	[k	¸	«	»	Ë	Û	ë	ü
C	NP	FS	,	<	L	\	l		¼	¼	Ì	Ü	ì	ü
D	CR	GS	-	=	M]	m	3	½	½	Í	Ý	í	ý
E	SO	RS	.	>	N	^	n	~	¾	¾	Î	Þ	î	þ
F	SI	US	/	?	O	_	o	DEL	¸	¸	Ï	ß	ï	ÿ

Multibyte Encoding Schemes

Multibyte encoding schemes are needed to support ideographic scripts used in Asian languages like Chinese or Japanese because these languages use thousands of characters. These schemes use either a fixed number of bytes to represent a character or a variable number of bytes per character.

Fixed-Width Multibyte Encoding Schemes In a fixed-width multibyte encoding scheme, each character is represented by a fixed number of n bytes, where n is greater than or equal to two.

Variable-Width Multibyte Encoding Schemes A variable-width encoding scheme uses one or more bytes to represent a single character. Some multibyte encoding schemes use certain bits to indicate the number of bytes that will represent a character. For example, if two bytes is the maximum number of bytes used to represent a character, the most significant bit can be toggled to indicate whether that byte is a single-byte character or the first byte of a double-byte character. In other schemes, control codes differentiate single-byte from double-byte characters. Another possibility is that a shift-out code is used to indicate that the subsequent bytes are double-byte characters until a shift-in code is encountered.

Oracle's Naming Convention for Character Sets

Oracle uses the following naming convention for character set names:

`<language_or_region><#_of_bits_representing_a_character><standard_name>[S | C]`

Note that UTF8 and UTFE are exceptions to this naming convention.

Some examples are:

- US7ASCII is the U.S. 7-bit ASCII character set
- WE8ISO8859P1 is the Western European 8-bit ISO 8859 Part 1 character set
- JA16SJIS is the Japanese 16-bit Shifted Japanese Industrial Standard character set

The optional "S" or "C" at the end of the character set name is used to differentiate character sets that can be used only on the server (S) or only on the client (C).

On Macintosh platforms, the server character set should always be used. The Macintosh client character sets are obsolete. On EBCDIC platforms, if available, the "S" version should be used on the server and the "C" version on the client.

Choosing an Oracle Database Character Set

Oracle uses the database character set for:

- Data stored in SQL CHAR datatypes (CHAR, VARCHAR2, CLOB, and LONG)
- Identifiers such as table names, column names, and PL/SQL variables
- Entering and storing SQL and PL/SQL program source

Consider the following questions when you choose an Oracle character set for the database:

- What languages does the database need to support?
- What are interoperability concerns with system resources and applications?
- What are the performance implications?
- What are the restrictions?

Several character sets may meet your current language requirements, but you should consider future language requirements as well. If you know that you will need to expand support in the future for different languages, picking a character set with a wider range now will prevent the need for migration later. The Oracle

character sets listed in [Appendix A, "Locale Data"](#) are named according to the languages and regions which are covered by a particular character set. In the case of regions covered, some character sets (for example, the ISO character sets) are also listed explicitly by language. You may want to see the actual characters that are encoded. Most character sets are based on national, international, or vendor product documentation, or are available in standards documents.

Interoperability with System Resources and Applications

While the database maintains and processes the actual character data, there are other resources that you must depend on from the operating system. For example, the operating system supplies fonts that correspond to the character set you have chosen. Input methods that support the desired languages and application software must also be compatible with a particular character set.

Ideally, a character set should be available on the operating system and is handled by your application to ensure seamless integration.

Character Set Conversion

If you choose a character set that is different from what is available on the operating system, the Oracle database can convert the operating system character set to the database character set. However, there is some character set conversion overhead, and you need to make sure that the operating system character set has an equivalent character repertoire to avoid data loss.

Character set conversions can sometimes cause data loss. For example, if you are converting from character set A to character set B, the destination character set B must have the same character set repertoire as A. Any characters that are not available in character set B will be converted to a replacement character, which is most often specified as a question mark, (?), or a linguistically related character. For example, ä (a with an umlaut) will be converted to a. If you have distributed environments, consider using character sets with similar character repertoires to avoid loss of data.

Character set conversion may require copying strings between buffers multiple times before the data reaches the client. Therefore, if possible, use the same character sets for the client and the server to optimize performance.

See Also: [Chapter 10, "Character Set Scanner Utility"](#)

Database Schemas

By default, the character datatypes `CHAR` and `VARCHAR2` are specified in bytes, not characters. Hence, the specification `CHAR (20)` in a table definition allows 20 bytes for storing character data.

This works well if the database character set uses a single-byte character encoding scheme because the number of characters will be the same as the number of bytes. If the database character set uses a multibyte character encoding scheme, there is no such correspondence. That is, the number of bytes no longer equals the number of characters since a character can consist of one or more bytes. Thus, column widths must be chosen with care to allow for the maximum possible number of bytes for a given number of characters. You can overcome this problem by switching to character semantics when defining the column size.

See Also: *Oracle9i Database Concepts* for more information about character semantics

Performance Implications

There can be different performance overheads in handling different encoding schemes, depending on the character set chosen. For best performance, you should try to choose a character set that avoids character set conversion and uses the most efficient encoding for the languages desired. Single-byte character sets are more optimal for performance than multibyte character sets, and they also are the most efficient in terms of space requirements. However, single-byte character sets limit how many languages you can use.

Restriction

ASCII-based character sets are supported only on ASCII-based platforms. Similarly, you can use an EBCDIC-based character set only on EBCDIC-based platforms.

The database character set is used to identify SQL and PL/SQL source code. In order to do this, it must have either EBCDIC or 7-bit ASCII as a subset, whichever is native to the platform. Therefore, it is not possible to use a fixed-width, multibyte character set as the database character set. Currently, this restriction applies only to the AL16UTF16 character set.

Choosing an Oracle NCHAR Character Set

In some cases, you may wish to choose an alternate character set for the database because:

- The properties of a different character encoding scheme may be more desirable for extensive character processing operations
- Programming in the alternate character set is easier

SQL `NCHAR` datatypes have been redefined to support Unicode data only. You can store the data in either UTF-8 or UTF-16 encodings.

See Also: [Chapter 5, "Supporting Multilingual Databases with Unicode"](#)

Restrictions on Character Sets Used to Express Names and Text

[Table 2–4](#) lists the restrictions on the character sets that can be used to express names and other text in Oracle.

Table 2–4 *Restrictions on Character Sets Used to Express Names and Text*

Name	Single-Byte or Fixed-Width	Variable Width	Comments
column names	Yes	Yes	
schema objects	Yes	Yes	
comments	Yes	Yes	
database link names	Yes	No	
database names	Yes	No	
filenames (datafile, log file, control file, initialization parameter file)	Yes	No	
instance names	Yes	No	
directory names	Yes	No	
keywords	Yes	No	Can be expressed in English ASCII or EBCDIC characters only
recovery manager filenames	Yes	No	

Table 2–4 Restrictions on Character Sets Used to Express Names and Text (Cont.)

Name	Single-Byte or Fixed-Width	Variable Width	Comments
rollback segment names	Yes	No	The <code>ROLLBACK_SEGMENTS</code> parameter does not support NLS
stored script names	Yes	Yes	
tablespace names	Yes	No	

For a list of supported string formats and character sets, including LOB data (LOB, BLOB, CLOB, and NCLOB), see [Table 2–6](#).

The character encoding scheme used by the database is defined at database creation as part of the `CREATE DATABASE` statement. All SQL `CHAR` datatype columns (`CHAR`, `CLOB`, `VARCHAR2`, and `LONG`), including columns in the data dictionary, have their data stored in the database character set. In addition, the choice of database character set determines which characters can name objects in the database. SQL `NCHAR` datatype columns (`NCHAR`, `NCLOB`, and `NVARCHAR2`) use the national character set.

After the database is created, the character set choices cannot be changed, with some exceptions, without re-creating the database. Hence, it is important to consider carefully which character sets to use. The database character set should always be a superset or equivalent of the client's operating system's native character set. The character sets used by client applications that access the database usually determine which superset is the best choice.

If all client applications use the same character set, then this is the normal choice for the database character set. When client applications use different character sets, the database character set should be a superset of all the client character sets. This ensures that every character is represented when converting from a client character set to the database character set.

When a client application operates with a terminal that uses a different character set, then the client application's characters must be converted to the database character set, and vice versa. This conversion is performed automatically, and is transparent to the client application, except that the number of bytes for a character string may be different in the client character set and the database character set. The character set used by the client application is defined by the `NLS_LANG` parameter.

Summary of Datatypes and Supported Encoding Schemes

[Table 2–5](#) lists the supported encoding schemes associated with different datatypes.

Table 2–5 *Supported Encoding Schemes for Datatypes*

Datatype	Single Byte	Multibyte Non-Unicode	Multibyte Unicode
CHAR	Yes	Yes	Yes
VARCHAR2	Yes	Yes	Yes
NCHAR	No	No	Yes
NVARCHAR2	No	No	Yes
BLOB	Yes	Yes	Yes
CLOB	Yes	Yes	Yes
LONG	Yes	Yes	Yes
NCLOB	No	No	Yes

[Table 2–6](#) lists the supported datatypes associated with Abstract Data Types (ADT).

Table 2–6 *Supported Datatypes for Abstract Datatypes*

Abstract Datatype	CHAR	NCHAR	BLOB	CLOB	NCLOB
Object	Yes	No	Yes	Yes	No
Collection	Yes	No	Yes	Yes	No

Note: BLOBs process characters as a series of byte sequences. The data is not subject to any NLS-sensitive operations.

Changing the Character Set After Database Creation

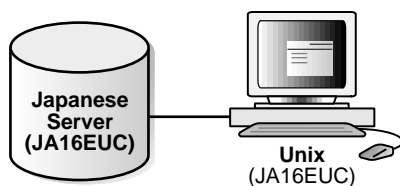
In some cases, you may wish to change the existing database character set. For example, you may find that the number of languages that need to be supported in your database have increased. In most cases, you will need to do a full export/import to properly convert all data to the new character set. However, if, and only if, the new character set is a strict superset of the current character set, it is possible to use the `ALTER DATABASE CHARACTER SET` statement to expedite the change in the database character set.

See Also: [Chapter 10, "Character Set Scanner Utility"](#) for more information about character set conversion

Monolingual Database Scenario

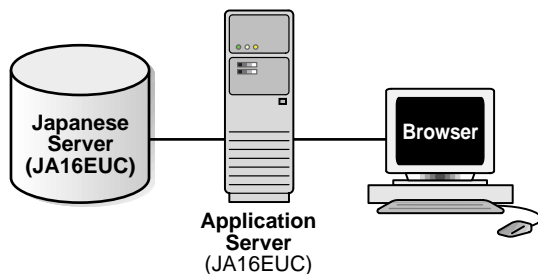
The simplest example of an NLS database setup is when both the client and the server run in the same language environment and use the same character encoding. This monolingual scenario has the advantage of fast response because the overhead associated with character set conversion is avoided. [Figure 2-2](#), illustrates this:

Figure 2-2 *Monolingual Database Scenario*



You can also use a multitier architecture, as illustrated in [Figure 2-3](#):

Figure 2-3 *Multitier Monolingual Database Scenario*



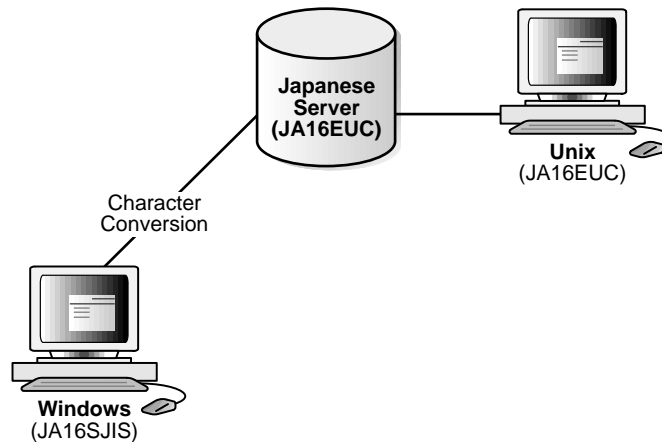
Character Set Conversion

You may need to convert character sets in a client/server computing environment because a client application resides on a different computer platform from that of the server, and both platforms do not use the same character encoding schemes.

Character data passed between client and server must be converted between the two encoding schemes. Character conversion occurs automatically and transparently via Oracle Net.

You can convert between any two character sets, as shown in [Figure 2-4](#):

Figure 2-4 Character Set Conversion



However, in cases where a target character set does not contain all characters in the source data, replacement characters are used. If, for example, a server uses US7ASCII and a German client WE8ISO8859P1, the German character ß is replaced with ? and ä is replaced with a.

Replacement characters may be defined for specific characters as part of a character set definition. When a specific replacement character is not defined, a default replacement character is used. To avoid the use of replacement characters when converting from client to database character set, the server character set should be a superset (or equivalent) of all the client character sets. In [Figure 2-2](#), the server's character set was not chosen wisely. If German data is expected to be stored on the server, a character set that supports German letters, such as WE8ISO8859P1, is needed for both the server and the client.

In some variable-width multibyte cases, character set conversion may introduce noticeable overhead. You need to carefully evaluate your situation and choose character sets to avoid conversion as much as possible. Having the appropriate character set for the database and the client will avoid the overhead of character conversion, as well as possible data loss.

Multilingual Database Scenarios

Note that some character sets support multiple languages. This is typical when the languages have related writing systems or scripts. For example, [Table 2-7](#) illustrates that WE8ISO8859P1 supports the following Western European languages:

Table 2-7 *WE8ISO8859P1 Example*

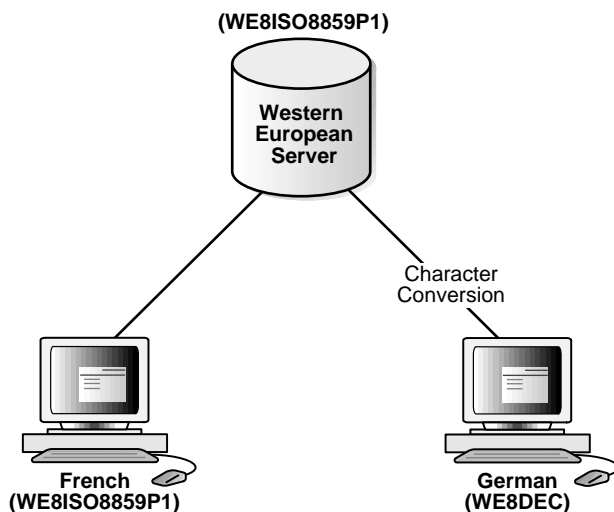
Catalan	Finnish	Icelandic	Portuguese
Danish	French	Italian	Spanish
Dutch	German	Norwegian	Swedish
English			

The reason WE8ISO8859P1 supports the languages above is because they are all based on a similar writing script. This situation is called *restricted* multilingual support. In this case, they are all Latin-based scripts.

Restricted Multilingual Support

In [Figure 2-5](#), both clients have access to the server's data, though the German client requires character conversion because it is using the WE8DEC character set.

Figure 2-5 *Restricted Multilingual Support*



Character conversion is necessary, but both French and German are Latin-based scripts, so you can use WE8ISO8859P1.

Unrestricted Multilingual Support

Often, *unrestricted* multilingual support is needed, and a universal character set such as Unicode is necessary as the server database character set. Unicode has two major encoding schemes: UTF-16 and UTF-8. UTF-16 is a two-byte fixed-width format; UTF-8 is a multibyte format with a variable width. The Oracle9i database provides support for UTF-8 as a database character set and both UTF-8 and UTF-16 as the national character set. This enhancement is transparent to clients who already provide support for multi-byte character sets.

Character set conversion between a UTF-8 database and any single-byte character set introduces very little overhead. Conversion between UTF-8 and any multibyte character set has some overhead but there is no conversion loss problem except that some multibyte character sets do not support user-defined characters during character set conversion to and from UTF-8.

See Also: [Appendix A, "Locale Data"](#)

[Figure 2-6](#), shows how a database can support many different languages. Here, Japanese, French, and German clients are all accessing the same database based on the Unicode character set. Note that each client accesses only data that it can process. If Japanese data were retrieved, modified, and stored by the German client, all Japanese characters would be lost during the character set conversion.

Figure 2–6 Unrestricted Multilingual Support Scenario

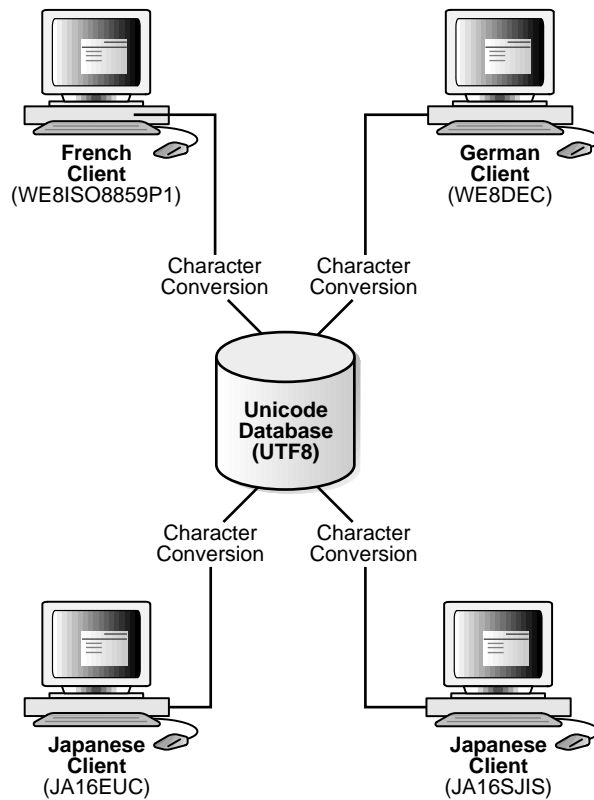


Figure 2–6 illustrates a Unicode solution for a client/server architecture. You can also use a multitier architecture, as illustrated in Figure 2–7.

Figure 2–7 *Multitier Unrestricted Multilingual Support Scenario*

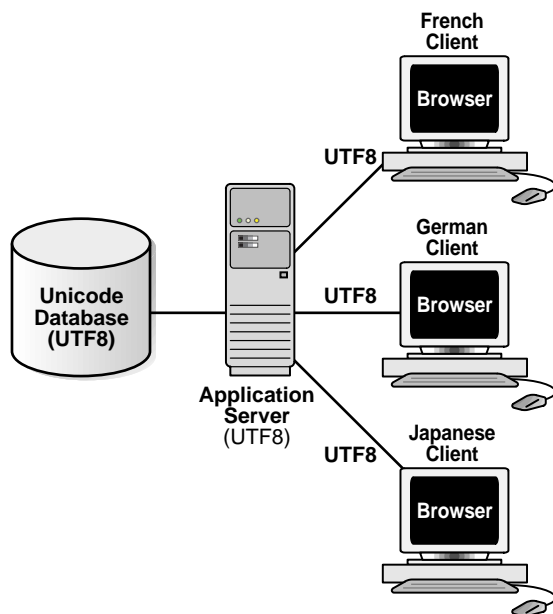


Figure 2–7 illustrates a multitier Unicode solution. Using this all-UTF8 architecture, you eliminate the need for character conversion.

See Also: [Chapter 5, "Supporting Multilingual Databases with Unicode"](#)

Setting Up a Globalization Support Environment

This chapter tells how to set up a globalization support environment. It includes the following topics:

- [Setting NLS Parameters](#)
- [Choosing a Locale with the NLS_LANG Initialization Parameter](#)
- [Checking NLS Parameters](#)
- [Date and Time Parameters](#)
- [Calendar Parameter](#)
- [Numeric Parameters](#)
- [Monetary Parameters](#)
- [Linguistic Sorting Parameters](#)
- [Character Set Parameter](#)

Setting NLS Parameters

NLS parameters determine the locale-specific behavior on both the client and the server. NLS parameters can be specified in the following ways:

- As **initialization parameters** on the server. You can include parameters in the initialization parameter file to specify a default session NLS environment. These settings have no effect on the client side; they control only the server's behavior. For example:

```
NLS_TERRITORY = "CZECH REPUBLIC"
```

In addition, `NLS_LENGTH_SEMANTICS` and `NLS_CONV_EXCP` can be set using the `ALTER SYSTEM` statement.

- As **environment variables** on the client. You can use NLS parameters to specify locale-dependent behavior for the client, and also to override the default values set for the session in the initialization parameter file. For example, on a UNIX system:

```
% setenv NLS_SORT FRENCH
```

- As **ALTER SESSION parameters**. NLS parameters that are set in an `ALTER SESSION` statement can be used to override the default values that are set for the session in the initialization parameter file or set by the client with environment variables.

```
ALTER SESSION SET NLS_SORT = FRENCH;
```

See Also: *Oracle9i SQL Reference* for more information about the `ALTER SESSION` statement

- As **SQL function parameters**. NLS parameters can be used explicitly to hardcode NLS behavior within a SQL function. Doing so will override the default values that are set for the session in the initialization parameter file, set for the client with environment variables, or set for the session by the `ALTER SESSION` statement. For example:

```
TO_CHAR(hiredate, 'DD/MON/YYYY', 'nls_date_language = FRENCH')
```

The database character set and the national character set are specified in the `CREATE DATABASE` statement.

See Also: *Oracle9i SQL Reference* for more information about the `CREATE DATABASE` statement

[Table 3–1](#) shows the precedence order when using NLS parameters. Higher priority settings will override lower priority settings. For example, a default value will have the lowest possible priority, and can be overridden by any other method. Explicitly setting an NLS parameter within a SQL function overrides all other settings — default, initialization parameter, environment variable, and `ALTER SESSION` parameters.

Table 3–1 *Parameter Settings and Their Priorities*

Highest Priority	
1	Explicitly set in SQL functions
2	Set by an <code>ALTER SESSION</code> statement
3	Set as an environment variable
4	Specified in the initialization parameter file
5	Default
Lowest Priority	

[Table 3–2](#) lists the NLS parameters available with the Oracle server.

Table 3–2 *NLS Parameters and their Scope*

Parameter	Description	Default	Scope: I= INIT.ORA, E= Environment Variable, A= Alter Session
NLS_CALENDAR	Calendar system	Gregorian	I, E, A
NLS_COMP	SQL, PL/SQL operator comparison	Binary	I, E, A
NLS_CREDIT	Credit accounting symbol	<code>NLS_TERRITORY</code>	-, E, -
NLS_CURRENCY	Local currency symbol	<code>NLS_TERRITORY</code>	I, E, A
NLS_DATE_FORMAT	Date format	<code>NLS_TERRITORY</code>	I, E, A
NLS_DATE_LANGUAGE	Language for day and month names	<code>NLS_LANGUAGE</code>	I, E, A
NLS_DEBIT	Debit accounting symbol	<code>NLS_TERRITORY</code>	-, E, -

Table 3–2 NLS Parameters and their Scope (Cont.)

NLS_ISO_CURRENCY	ISO international currency symbol	NLS_TERRITORY	I, E, A
NLS_LANG	Language, territory, character set	AMERICAN_ AMERICA. US7ASCII	-, E, -
NLS_LANGUAGE	Language	NLS_LANG	I, -, A
NLS_LENGTH_SEMANTICS	How strings are treated	Byte	I, -, A
NLS_LIST_SEPARATOR	Character separating items in a list	NLS_TERRITORY	-, E, -
NLS_MONETARY_CHARACTERS	Monetary symbol for dollar and cents (or their equivalents)	NLS_TERRITORY	-, E, -
NLS_NCHAR_CONV_EXCP	Reports data loss during a character type conversion		-, E, -
NLS_NUMERIC_CHARACTERS	Decimal character and group separator	NLS_TERRITORY	I, E, A
NLS_SORT	Character Sort Sequence	NLS_LANGUAGE	I, E, A
NLS_TERRITORY	Territory	NLS_LANG	I, -, A
NLS_TIMESTAMP_FORMAT	Timestamp	NLS_TERRITORY	I, E, A
NLS_TIMESTAMP_TZ_FORMAT	Timestamp with Timezone	NLS_TERRITORY	I, E, A
NLS_DUAL_CURRENCY	Dual currency symbol	NLS_TERRITORY	I, E, A

Choosing a Locale with the NLS_LANG Initialization Parameter

A **locale** is a linguistic and cultural environment in which a system or program is running. Setting the NLS_LANG parameter is the simplest way to specify locale behavior. It sets the language and territory used by the client application. It also sets the character set of the client, which is the character set of data entered or displayed by a client program.

The NLS_LANG parameter has three components (language, territory, and character set) in the form:

```
NLS_LANG = language_territory.charset
```

Each component controls the operation of a subset of NLS features:

<i>language</i>	Specifies conventions such as the language used for Oracle messages, sorting, day names, and month names. Each supported language has a unique name; for example, AMERICAN, FRENCH, or GERMAN. The language argument specifies default values for the territory and character set arguments. If language is not specified, the value defaults to AMERICAN. For a complete list of languages, see Appendix A, "Locale Data" .
<i>territory</i>	Specifies conventions such as the default date, monetary, and numeric formats. Each supported territory has a unique name; for example, AMERICA, FRANCE, or CANADA. If territory is not specified, the value defaults from the language value. For a complete list of territories, see Appendix A, "Locale Data" .
<i>charset</i>	Specifies the character set used by the client application (normally that of the user's terminal). Each supported character set has a unique acronym, for example, US7ASCII, WE8ISO8859P1, WE8DEC, WE8EBCDIC500, or JA16EUC. Each language has a default character set associated with it. For a complete list of character sets, see Appendix A, "Locale Data" .

Note: All components of the NLS_LANG definition are optional; any item left out will default. If you specify territory or charset, you *must* include the preceding delimiter [underscore (_) for territory, period (.) for charset]. Otherwise, the value will be parsed as a language name.

The three arguments of NLS_LANG can be specified in many combinations, as in the following examples:

```
NLS_LANG = AMERICAN_AMERICA.US7ASCII
```

or

```
NLS_LANG = FRENCH_CANADA.WE8DEC
```

or

```
NLS_LANG = JAPANESE_JAPAN.JA16EUC
```

Note that illogical combinations can be set but will not work properly. For example, the following specification tries to support Japanese by using a Western European character set:

```
NLS_LANG = JAPANESE_JAPAN.WE8DEC
```

Because WE8DEC does not support any Japanese characters, you would be unable to store Japanese data.

Specifying NLS_LANG as an Environment Variable

You can set NLS_LANG as an environment variable at the command line. For example, on UNIX, you can specify the value of NLS_LANG by entering a statement similar to the following:

```
% setenv NLS_LANG FRENCH_FRANCE.WE8DEC
```

NLS_LANG Examples

Because NLS_LANG is an environment variable, it is read by the client application at startup time. The client communicates the information defined by NLS_LANG to the server when it connects to the database server.

The following examples show how date and number formats are affected by the NLS_LANG parameter.

```
% setenv NLS_LANG American_America.WE8ISO8859P1
SQL> SELECT ename, hiredate, ROUND(sal/12,2) sal FROM emp;
ENAME                HIREDATE                SAL
-----
Clark                09-DEC-88                4195.83
Miller               23-MAR-92                4366.67
Strauß               01-APR-95                3795.87
```

If NLS_LANG is set with the language as French, the territory as France, and the character set as Western European 8-bit ISO 8859-1, then the same query returns the following information:

```
% setenv NLS_LANG French_France.WE8ISO8859P1
SQL> SELECT ename, hiredate, ROUND(sal/12,2) sal FROM emp;
ENAME                HIREDATE                SAL
-----
Clark                09/12/88                4195,83
Miller               23/03/92                4366,67
Strauß               01/04/95                3795,87
```

Overriding Language and Territory Specifications

The NLS_LANG parameter sets the language and territory environment used by both the server session (for example, SQL command execution) and the client application (for example, display formatting in Oracle tools). Using this parameter

ensures that the language environments of both database and client application are automatically the same.

The language and territory components of the `NLS_LANG` parameter set the default values for the other NLS parameters, such as date format, numeric characters, and linguistic sorting. Each of these detailed parameters can be set in the client environment to customize the language and territory values.

Note that NLS parameters in the client environment are ignored if `NLS_LANG` is not set.

If the `NLS_LANG` parameter is not set, the server session environment remains initialized with values of `NLS_LANGUAGE`, `NLS_TERRITORY`, and other NLS instance parameters from the initialization parameter file. You can modify these parameters and restart the instance to change the defaults.

You might want to modify your NLS environment dynamically during the session. To do so, you can use `NLS_LANGUAGE`, `NLS_TERRITORY` and other NLS parameters in the `ALTER SESSION` statement.

The `ALTER SESSION` statement modifies only the session environment. The local client NLS environment is not modified, unless the client explicitly retrieves the new settings and modifies its local environment. SQL*Plus is an example of an application that retrieves new settings; Oracle Developer is an example of an application that does not retrieve new settings.

NLS Database Parameters

When a new database is created during the execution of `CREATE DATABASE` statement, the NLS database environment is established. The current NLS instance parameters, as defined by the initialization parameter file, are stored in the data dictionary along with the database and national character sets.

Checking NLS Parameters

You can find the values for NLS parameters in some views or by using an OCI function call.

NLS Views

Applications can check the current session, instance and database NLS parameters by querying the following data dictionary views:

- `NLS_SESSION_PARAMETERS` shows the current NLS parameters of the session querying the view.
- `NLS_INSTANCE_PARAMETERS` shows the current NLS parameters of the instance, that is, NLS parameters read from the initialization parameter file at instance startup. The view shows only parameters that were explicitly set.
- `NLS_DATABASE_PARAMETERS` shows the current NLS parameters of the database, including the database character set.
- `V$NLS_VALID_VALUES` can be used to see which language, territory, linguistic and character set definitions are supported by the server.

See Also: *Oracle9i Database Reference*

OCI Functions

User applications can query client NLS settings with the `OCI-NLSGetInfo` function.

See Also: [Chapter 8, "OCI Programming"](#) for the description of `OCI-NLSGetInfo`

Language and Territory Parameters

`NLS_LANGUAGE` and `NLS_TERRITORY` parameters are general NLS parameters that describe NLS behavior of locale-dependent operations.

`NLS_LANGUAGE`

Parameter type:	String
Parameter scope:	Initialization Parameter and <code>ALTER SESSION</code>
Default value:	Derived from <code>NLS_LANG</code>
Range of values:	Any valid language name

`NLS_LANGUAGE` specifies the default conventions for the following session characteristics:

- Language for server messages
- Language for day and month names and their abbreviations (specified in the SQL functions `TO_CHAR` and `TO_DATE`)
- Symbols for equivalents of AM, PM, AD, and BC. (A.M., P.M., A.D., and B.C. are only valid if `NLS_LANGUAGE` is set to `AMERICAN`.)

- Default sorting sequence for character data when `ORDER BY` is specified. (`GROUP BY` uses a binary sort, unless `ORDER BY` is specified.)
- Writing direction
- Affirmative and negative response strings

The value specified for `NLS_LANGUAGE` in the initialization parameter file is the default for all sessions in that instance. For example, to specify the default session language as French, the parameter should be set as follows:

```
NLS_LANGUAGE = FRENCH
```

In this case, the server message

```
ORA-00942: table or view does not exist
```

will appear as

```
ORA-00942: table ou vue inexistante
```

Messages used by the server are stored in binary-format files that are placed in the `$ORACLE_HOME/product_name/mesg` directory, or the equivalent. Multiple versions of these files can exist, one for each supported language, using the filename convention:

```
<product_id><language_abbrev>.MSB
```

For example, the file containing the server messages in French is called `ORAF.MSB`, with `F` being the language abbreviation for French.

Messages are stored in these files in one specific character set, depending on the language and operating system. If this is different from the database character set, message text is automatically converted to the database character set. If necessary, it will be further converted to the client character set if it is different from the database character set. Hence, messages will be displayed correctly at the user's terminal, subject to the limitations of character set conversion.

The default value of `NLS_LANGUAGE` may be operating system specific. You can alter the `NLS_LANGUAGE` parameter by changing the value in the initialization parameter file and then restarting the instance.

For more information on the default value, see your operating system-specific Oracle documentation.

The following examples show behavior before and after setting `NLS_LANGUAGE`.

```
ALTER SESSION SET NLS_LANGUAGE=Italian;
```

```
SELECT ename, hiredate, ROUND(sal/12,2) sal FROM emp;
ENAME      HIREDATE      SAL
-----      -
Clark      09-Dic-88      4195.83
Miller     23-Mar-87      4366.67
Strauß     01-Apr-95      3795.87

SQL> ALTER SESSION SET NLS_LANGUAGE=German;
SQL> SELECT ename, hiredate, ROUND(sal/12,2) sal FROM emp;
ENAME      HIREDATE      SAL
-----      -
Clark      09-DEZ-88      4195.83
Miller     23-MÄR-87      4366.67
Strauß     01-APR-95      3795.87
```

NLS_LENGTH_SEMANTICS

- Parameter type:** String
- Parameter scope:** Dynamic, Initialization Parameter, ALTER SESSION, and ALTER SYSTEM
- Default value:** BYTE
- Range of values:** BYTE | CHAR

NLS_LENGTH_SEMANTICS enables you to create CHAR, VARCHAR2, and LONG columns using either byte or character length semantics. NCHAR, NVARCHAR2, CLOB, and NCLOB columns are always character-based. Existing columns are not affected.

You may be required to use byte semantics in order to maintain compatibility with existing applications.

NLS_LENGTH_SEMANTICS does not apply to tables in SYS and SYSTEM. The data dictionary will always use byte semantics.

NLS_TERRITORY

- Parameter type:** String
- Parameter scope:** Initialization Parameter and ALTER SESSION
- Default value:** Derived from NLS_LANG
- Range of values:** Any valid territory name

NLS_TERRITORY specifies the conventions for the following default date and numeric formatting characteristics:

- Date format
- Decimal character and group separator
- Local currency symbol
- ISO currency symbol
- Dual currency symbol
- Week start day
- Credit and debit symbol
- ISO week flag
- List separator

The value specified for `NLS_TERRITORY` in the initialization parameter file is the default for the instance. For example, to specify the default as France, the parameter should be set as follows:

```
NLS_TERRITORY = FRANCE
```

In this case, numbers would be formatted using a comma as the decimal character.

You can alter the `NLS_TERRITORY` parameter by changing the value in the initialization parameter file and then restarting the instance. The default value of `NLS_TERRITORY` can be operating system specific.

If `NLS_LANG` is specified in the client environment, the value in the initialization parameter file is overridden at connection time.

The territory can be modified dynamically during the session by specifying the new `NLS_TERRITORY` value in an `ALTER SESSION` statement. Modification of `NLS_TERRITORY` resets all derived NLS session parameters to default values for the new territory.

To change the territory dynamically to France, issue the following statement:

```
ALTER SESSION SET NLS_TERRITORY=France;
```

The following examples show behavior before and after setting `NLS_TERRITORY`:

```
SQL> DESCRIBE SalaryTable;
Name                               Null?      TYPE
-----
SALARY                             NUMBER
```

```
SQL> column SALARY format L999,999.99;
SQL> SELECT * from SalaryTable;
           SALARY
-----
          $100,000.00
          $150,000.00

ALTER SESSION SET NLS_TERRITORY = Germany;
Session altered.

SQL> SELECT * from SalaryTable;
           SALARY
-----
        DM100,000.00
        DM150,000.00

ALTER SESSION SET NLS_LANGUAGE = German;
Sitzung wurde geändert.

SQL> SELECT * from SalaryTable;
           SALARY
-----
        DM100,000.00
        DM150,000.00

ALTER SESSION SET NLS_TERRITORY = France;
Sitzung wurde geändert.

SQL> SELECT * from SalaryTable;
           SALARY
-----
        F100,000.00
        F150,000.00
```

Note that the symbol for currency units changed, but no monetary conversion calculations were performed. The numeric characters did not change because they were hardcoded by the SQL*Plus statement.

ALTER SESSION

The default values for language and territory can be overridden during a session by using the ALTER SESSION statement. For example:

```
% setenv NLS_LANG Italian_Italy.WE8DEC
```

```
SQL> SELECT ename, hiredate, ROUND(sal/12,2) sal FROM emp;
```

ENAME	HIREDATE	SAL
-----	-----	---
Clark	09-Dic-88	4195,83
Miller	23-Mar-87	4366,67
Strauß	01-Apr-95	3795,87

```
ALTER SESSION SET NLS_LANGUAGE = German
NLS_DATE_FORMAT = 'DD.MON.YY'
NLS_NUMERIC_CHARACTERS = '.,';
```

```
SQL> SELECT ename, hiredate, ROUND(sal/12,2) sal FROM emp;
```

ENAME	HIREDATE	SAL
-----	-----	---
Clark	09.DEZ.88	4195.83
Miller	23.MÄR.87	4366.67
Strauß	01.APR.95	3795.87

This feature implicitly determines the language environment of the database for each session. An `ALTER SESSION` statement is automatically executed when a session connects to a database to set the values of the database parameters `NLS_LANGUAGE` and `NLS_TERRITORY` to those specified by the language and territory arguments of `NLS_LANG`. If `NLS_LANG` is not defined, no implicit `ALTER SESSION` statement is executed.

When `NLS_LANG` is defined, the implicit `ALTER SESSION` is executed for all instances to which the session connects, for both direct and indirect connections. If the values of NLS parameters are changed explicitly with `ALTER SESSION` during a session, the changes are propagated to all instances to which that user session is connected.

Messages and Text

All messages and text should be in the same language. For example, when running an Oracle Developer application, messages and boilerplate text seen by the user originate from three sources:

- Messages from the server
- Messages and boilerplate text generated by Oracle Forms
- Messages and boilerplate text defined as part of the application

The application is responsible for meeting the last requirement. NLS takes care of the other two.

Date and Time Parameters

Oracle enables you to control many aspects of date and time display.

Date Formats

Many different date formats are used throughout the world. Some typical ones are shown in [Table 3–3](#).

Table 3–3 *Date Formats*

Country	Description	Example
Estonia	dd.mm.yyyy	28.02.1998
Germany	dd-mm-rr	28-02-98
Japan	rr-mm-dd	98-02-28
UK	dd-mon-rr	28-Feb-98
US	dd-mon-rr	28-Feb-98

NLS_DATE_FORMAT

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Default format for a particular territory
Range of values:	Any valid date format mask

This parameter defines the default date format to use with the TO_CHAR and TO_DATE functions. The default value of this parameter is determined by NLS_TERRITORY. The value of this parameter can be any valid date format mask, and the value must be surrounded by quotation marks. For example:

```
NLS_DATE_FORMAT = "MM/DD/YYYY"
```

To add string literals to the date format, enclose the string literal with double quotes. Note that every special character (such as the double quote) must be preceded with an escape character. The entire expression must be surrounded with single quotes. For example:

```
NLS_DATE_FORMAT = '\\"Today\\'s date\\" MM/DD/YYYY'
```

As another example, to set the default date format to display Roman numerals for months, you would include the following line in the initialization file:

```
NLS_DATE_FORMAT = "DD RM YYYY"
```

With such a default date format, the following `SELECT` statement would return the month using Roman numerals (assuming today's date is February 12, 1997):

```
SELECT TO_CHAR(SYSDATE) CURRDATE
       FROM DUAL;
CURRDATE
-----
12 II 1997
```

The value of this parameter is stored in the internal date format. Each format element occupies two bytes, and each string occupies the number of bytes in the string plus a terminator byte. Also, the entire format mask has a two-byte terminator. For example, "MM/DD/YY" occupies 12 bytes internally because there are three format elements, two one-byte strings (the two slashes), and the two-byte terminator for the format mask. The format for the value of this parameter cannot exceed 24 bytes.

Note: The applications you design may need to allow for a variable-length default date format. Also, the parameter value must be surrounded by double quotes. Single quotes are interpreted as part of the format mask.

You can alter the default value of `NLS_DATE_FORMAT` by changing its value in the initialization file and then restarting the instance, and you can alter the value during a session using an `ALTER SESSION SET NLS_DATE_FORMAT` statement.

See Also: *Oracle9i SQL Reference* for more information about date format elements

Date Formats and Partition Bound Expressions Partition bound expressions for a date column must specify a date using a format that requires the month, day, and 4-digit year to be fully specified. For example, the date format MM-DD-YYYY requires that the month, day, and 4-digit year are fully specified. In contrast, the date format DD-MON-YY (11-jan-97, for example) is invalid because it relies on the current date for the century.

Use `TO_DATE` to specify a date format which requires the full specification of month, day, and 4-digit year. For example:

```
TO_DATE('11-jan-1997', 'dd-mon-yyyy')
```

If the default date format, specified by `NLS_DATE_FORMAT`, of your session does not support specification of a date independent of the current century (if your default date format is MM-DD-YY for example), then you must take one of the following actions:

- Use `TO_DATE` to express the date in a format that requires you to fully specify the day, month, and 4-digit year.
- Change the value of `NLS_DATE_FORMAT` for the session to support the specification of dates in a format which requires you to fully specify the day, month, and 4-digit year.

See Also: *Oracle9i SQL Reference* for more information about using `TO_DATE`

NLS_DATE_LANGUAGE

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Derived from <code>NLS_LANGUAGE</code>
Range of values:	Any valid language name

This parameter specifies the language for the spelling of day and month names by the functions `TO_CHAR` and `TO_DATE`, overriding that specified implicitly by `NLS_LANGUAGE`. `NLS_DATE_LANGUAGE` has the same syntax as the `NLS_LANGUAGE` parameter, and all supported languages are valid values. For example, to specify the date language as French, the parameter should be set as follows:

```
NLS_DATE_LANGUAGE = FRENCH
```

In this case, the query

```
SELECT TO_CHAR(SYSDATE, 'Day:Dd Month yyyy')
FROM DUAL;
```

returns

```
Mercredi:12 Février 1997
```

Month and day name abbreviations are also in the specified language. For example:

```
SELECT TO_CHAR(SYSDATE, 'Dy:dd Mon yyyy')
FROM DUAL;
```

```
Me:12 Fév 1997
```

The default date format also uses the language-specific month name abbreviations. For example, if the default date format is DD-MON-YYYY, then the above date can be inserted as follows:

```
INSERT INTO tablename VALUES ('12-Fév-1997');
```

The abbreviations for AM, PM, AD, and BC are also returned in the language specified by NLS_DATE_LANGUAGE. Note that numbers spelled using the TO_CHAR function always use English spellings. For example:

```
SELECT TO_CHAR(TO_DATE('12-Fév'), 'Day: ddsph Month')
FROM DUAL;
```

returns:

```
Mercredi: twelfth Février
```

You can alter the default value of NLS_DATE_LANGUAGE by changing its value in the initialization parameter file and then restarting the instance. You can alter the value during a session using an ALTER SESSION SET NLS_DATE_LANGUAGE statement.

Time Formats

Many different time formats are used throughout the world. Some typical time formats are shown in [Table 3-4](#).

Table 3–4 Time Formats

Country	Description	Example
Estonia	hh24:mi:ss	13:50:23
Germany	hh24:mi:ss	13:50:23
Japan	hh24:mi:ss	13:50:23
UK	hh24:mi:ss	13:50:23
US	hh:mi:ssx ^{ff} am	1:50:23.555 PM

NLS_TIMESTAMP_FORMAT

- Parameter type:** String
- Parameter scope:** Dynamic, Initialization Parameter, Environment Variable, and ALTER SESSION
- Default value:** Derived from NLS_TERRITORY
- Range of values:** Any valid datetime format mask

NLS_TIMESTAMP_FORMAT defines the default timestamp format to use with TO_CHAR and TO_TIMESTAMP functions. The value must be surrounded by quotation marks as follows:

```
NLS_TIMESTAMP_FORMAT = 'YYYY-MM-DD HH:MI:SS.FF'
```

An example is:

```
TO_TIMESTAMP('11-nov-2000 01:00:00.336', 'dd-mon-yyyy hh:mi:ss.ff')
```

You can specify the value of NLS_TIMESTAMP_FORMAT by setting it in the initialization parameter file. You can specify its value for a client as a client environment variable.

You can also alter the value of NLS_TIMESTAMP_FORMAT by changing its value in the initialization parameter file and then restarting the instance. To alter the value during a session, use the ALTER SESSION SET statement.

NLS_TIMESTAMP_TZ_FORMAT

- Parameter type:** String

Parameter scope:	Dynamic, Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Derived from NLS_TERRITORY
Range of values:	Any valid datetime format mask

NLS_TIMESTAMP_TZ_FORMAT defines the default timestamp with time zone format to use with TO_CHAR and TO_TIMESTAMP_TZ functions. The value must be surrounded by quotation marks as follows:

```
NLS_TIMESTAMP_TZ_FORMAT = 'YYYY-MM-DD HH:MI:SS.FF TZH:TZM'
```

An example is:

```
TO_TIMESTAMP_TZ('2000-08-20, 05:00:00.55 America/Los_Angeles', 'yyyy-mm-dd
hh:mi:ss.ff TZR')
```

You can specify the value of NLS_TIMESTAMP_TZ_FORMAT by setting it in the initialization parameter file. You can specify its value for a client as a client environment variable.

You can also alter the value of NLS_TIMESTAMP_TZ_FORMAT by changing its value in the initialization parameter file and then restarting the instance. To alter the value during a session, use the ALTER SESSION SET statement.

See Also: *Oracle9i SQL Reference* for more information about date format elements and time zone formats

Time Zone Parameters for Databases You can create a database with a specific time zone by specifying:

- A displacement from UTC (Coordinated Universal Time, formerly Greenwich Mean Time). The following example sets the time zone of the database to Pacific Standard time (eight hours behind UTC):

```
CREATE DATABASE ... SET TIME_ZONE = '-08:00 ';
```

- A time zone region. The following example also sets the time zone of the database to Pacific Standard time in the United States:

```
CREATE DATABASE ... SET TIME_ZONE = 'PST ';
```

To see a listing of valid region names, query the V\$TIMEZONE_NAMES view.

The database time zone is relevant only for TIMESTAMP WITH LOCAL TIME ZONE columns. Oracle normalizes all TIMESTAMP WITH LOCAL TIME ZONE data to the

time zone of the database when the data is stored on disk. If you do not specify the `SET TIME_ZONE` clause, then Oracle uses the operating system's time zone of the server. If the operating system's time zone is not a valid Oracle time zone, the database time zone defaults to UTC.

After the database has been created, you can change the time zone by issuing the `ALTER DATABASE SET TIME_ZONE` statement and then shutting down and starting up the database. The following example sets the time zone of the database to London time:

```
ALTER DATABASE SET TIME_ZONE = 'Europe/London ';
```

To find out the time zone of a database, use the `DBTIMEZONE` function as shown in the following example:

```
SELECT dbtimezone FROM dual;
```

```
DBTIME
-----
-08:00
```

Time Zone Parameters for Sessions You can change the time zone parameter of a user session by issuing an `ALTER SESSION` statement:

- **O/S Local Time Zone**

```
ALTER SESSION SET TIME_ZONE = local;
```

- **Database Time Zone**

```
ALTER SESSION SET TIME_ZONE = DBTIMEZONE;
```

- **An absolute time difference**

```
ALTER SESSION SET TIME_ZONE = '-05:00';
```

- **A named region**

```
ALTER SESSION SET TIME_ZONE = 'America/New_York';
```

You can use the environment variable `ORA_SDTZ` to set the default client session time zone. This variable takes input like `DB_TZ`, `OS_TZ`, time zone region or numerical time zone offset. If `ORA_SDTZ` is set to `DB_TZ`, the session time zone will be the same as the database time zone. If it is set to `OS_TZ`, the session time zone will be same as the operating system's time zone. If `ORA_SDTZ` is set to an invalid Oracle time zone, Oracle uses the operating system's time zone as default session

time zone. If the operating system's time zone is not a valid Oracle time zone, the session time zone defaults to UTC. To find out the time zone of a user session, use the `SESSIONTIMEZONE` function as shown in the following example:

```
SELECT sessiontimezone FROM dual;
SESSIONTIMEZONE
-----
-08:00
```

See Also: [Chapter 12, "Customizing Locale Data"](#)

Calendar Parameter

Oracle allows you to control calendar-related items through the use of parameters.

Calendar Formats

The type of calendar information stored for each territory is as follows:

- [First Day of the Week](#)
- [First Calendar Week of the Year](#)
- [Number of Days and Months in a Year](#)
- [First Year of Era](#)

First Day of the Week

Some cultures consider Sunday to be the first day of the week. Others consider Monday to be the first day of the week. A German calendar starts with Monday.

Table 3–5 *First Day of the Week*

März 1998						
Mo	Di	Mi	Do	Fr	Sa	So
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

First Calendar Week of the Year

Many countries, Germany, for example, use weeks for scheduling, planning, and bookkeeping. Oracle supports this convention.

In the ISO standard, the year relating to an ISO week number can be different from the calendar year. For example, 1st Jan 1988 is in ISO week number 53 of 1987. A week always starts on a Monday and ends on a Sunday.

- If January 1 falls on a Friday, Saturday, or Sunday, then the week including January 1 is the last week of the previous year, because most of the days in the week belong to the previous year.
- If January 1 falls on a Monday, Tuesday, Wednesday, or Thursday, then the week is the first week of the new year, because most of the days in the week belong to the new year.

To support the ISO standard, the IW format element is provided that returns the ISO week number.

A typical example with four or more days in the first week is:

Table 3–6 Day of the Week Example 1

January 1998						
Mo	Tu	We	Th	Fr	Sa	Su
			1	2	3	4
						<= 1st week of 1998
5	6	7	8	9	10	11
						<= 2nd week of 1998
12	13	14	15	16	17	18
						<= 3rd week of 1998
19	20	21	22	23	24	25
						<= 4th week of 1998
26	27	28	29	30	31	
						<= 5th week of 1998

A typical example with three or fewer days in the first week is:

Table 3–7 Day of the Week Example 2

January 1999						
Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
						<= 53rd week of 1998
4	5	6	7	8	9	10
						<= 1st week of 1999
11	12	13	14	15	16	17
						<= 2nd week of 1999

Table 3–7 Day of the Week Example 2

January				1999				
18	19	20	21	22	23	24	<= 3rd week of 1999	
25	26	27	28	29	30	31	<= 4th week of 1999	

Number of Days and Months in a Year

Oracle supports six calendar systems in addition to the default Gregorian:

- Japanese Imperial—uses the same number of months and days as Gregorian, but the year starts with the beginning of each Imperial Era
- ROC Official—uses the same number of months and days as Gregorian, but the year starts with the founding of the Republic of China
- Persian—has 12 months of equal length
- Thai Buddha—uses a Buddhist calendar
- Arabic Hijrah—has 12 months with 354 or 355 days
- English Hijrah—has 12 months with 354 or 355 days

First Year of Era

The Islamic calendar starts from the year of the Hegira. The Japanese Imperial calendar starts from the beginning of an Emperor's reign. For example, 1998 is the tenth year of the Heisei era. It should be noted, however, that the Gregorian system is also widely understood in Japan, so both 98 and Heisei 10 can be used to represent 1998.

NLS_CALENDAR

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Gregorian
Range of values:	Any valid calendar format name

Many different calendar systems are in use throughout the world. NLS_CALENDAR specifies which calendar system Oracle uses.

NLS_CALENDAR can have one of the following values:

- Arabic Hijrah
- English Hijrah
- Gregorian
- Japanese Imperial
- Persian
- ROC Official (Republic of China)
- Thai Buddha

For example, if `NLS_CALENDAR` is set to "Japanese Imperial", the date format is `E YY-MM-DD`, and the date is May 15, 1997, then the `SYSDATE` is displayed as follows:

```
SELECT SYSDATE FROM DUAL;  
SYSDATE  
-----  
H 09-05-15
```

Numeric Parameters

Oracle allows you to control how numbers appear.

Numeric Formats

The database must know the number-formatting convention used in each session to interpret numeric strings correctly. For example, the database needs to know whether numbers are entered with a period or a comma as the decimal character (234.00 or 234,00). Similarly, the application needs to be able to display numeric information in the format expected at the client site.

Some typical ones are shown in [Table 3–8](#).

Table 3–8 *Numeric Formats*

Country	Example Numeric Formats
Estonia	1 234 567,89
Germany	1.234.567,89
Japan	1,234,567.89
UK	1,234,567.89
US	1,234,567.89

NLS_NUMERIC_CHARACTERS

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Default decimal character and group separator for a particular territory
Range of values:	Any two valid numeric characters

This parameter specifies the decimal character and grouping separator, overriding those defined implicitly by NLS_TERRITORY. The group separator is the character that separates integer groups (that is, the thousands, millions, billions, and so on). The decimal character separates the integer and decimal parts of a number.

Any character can be the decimal or group separator. The two characters specified must be single-byte, and both characters must be different from each other. The characters cannot be any numeric character or any of the following characters: plus (+), hyphen (-), less than sign (<), greater than sign (>).

The characters are specified in the following format:

```
NLS_NUMERIC_CHARACTERS = "<decimal_character><group_separator>"
```

The grouping separator is the character returned by the number format mask G. For example, to set the decimal character to a comma and the grouping separator to a period, the parameter should be set as follows:

```
NLS_NUMERIC_CHARACTERS = ",."
```

Both characters are single byte and must be different. Either can be a space.

SQL statements can include numbers represented as numeric or text literals. Numeric literals are not enclosed in quotes. They are part of the SQL language syntax and always use a dot as the decimal separator and never contain a group separator. Text literals are enclosed in single quotes. They are implicitly or explicitly converted to numbers, if required, according to the current NLS settings. For example, in the following statement:

```
INSERT INTO SIZES (ITEMID, WIDTH, HEIGHT, QUANTITY)
VALUES (618, '45,5', 27.86, TO_NUMBER('1.234','9G999'));
```

618 and 27.86 are numeric literals. The text literal '45,5' is implicitly converted to the number 45.5 (assuming that WIDTH is a NUMBER column). The text literal '1.234' is

explicitly converted to a number 1234. This statement is valid only if NLS_NUMERIC_CHARACTERS is set to ",.".

You can alter the default value of NLS_NUMERIC_CHARACTERS in either of these ways:

- Change the value of NLS_NUMERIC_CHARACTERS in the initialization parameter file and then restart the instance.
- Use the ALTER SESSION SET NLS_NUMERIC_CHARACTERS statement to change the parameter's value during a session.

Monetary Parameters

Oracle allows you to control how currency and financial symbols appear.

Currency Formats

Many different currency formats are used throughout the world. Some typical ones are shown in [Table 3-9](#).

Table 3-9 *Currency Format Examples*

Country	Example
Estonia	1 234,56 kr
Germany	1.234,56 DM
Japan	¥1,234.56
UK	£1,234.56
US	\$1,234.56

NLS_CURRENCY

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Default local currency symbol for a particular territory
Range of values:	Any valid currency symbol string

This parameter specifies the character string returned by the number format mask L, the local currency symbol, overriding that defined implicitly by NLS_

TERRITORY. For example, to set the local currency symbol to "Dfl " (including a space), the parameter should be set as follows:

```
NLS_CURRENCY = "Dfl "
```

In this case, the query

```
SELECT TO_CHAR(TOTAL, 'L099G999D99') "TOTAL"  
FROM ORDERS WHERE CUSTNO = 586;
```

returns

```
TOTAL  
-----  
Dfl 12.673,49
```

You can alter the default value of NLS_CURRENCY by changing its value in the initialization parameter file and then restarting the instance. You can alter its value during a session using an ALTER SESSION SET NLS_CURRENCY statement.

NLS_ISO_CURRENCY

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Derived from NLS_TERRITORY
Range of values:	Any valid territory name

This parameter specifies the character string returned by the number format mask C, the ISO currency symbol, overriding the value defined implicitly by NLS_TERRITORY.

Local currency symbols can be ambiguous. For example, a dollar sign (\$) can refer to US dollars or Australian dollars. ISO Specification 4217 1987-07-15 defines unique "international" currency symbols for the currencies of specific territories or countries.

For example, the ISO currency symbol for the US Dollar is USD. For the Australian Dollar, it is AUD. To specify the ISO currency symbol, the corresponding territory name is used.

NLS_ISO_CURRENCY has the same syntax as the NLS_TERRITORY parameter, and all supported territories are valid values. For example, to specify the ISO currency symbol for France, the parameter should be set as follows:

```
NLS_ISO_CURRENCY = FRANCE
```

In this case, the query

```
SELECT TO_CHAR(TOTAL, 'C099G999D99') "TOTAL"
FROM orders WHERE custno = 586;
```

returns

```
TOTAL
-----
FRF12.673,49
```

You can alter the default value of NLS_ISO_CURRENCY by changing its value in the initialization parameter file and then restarting the instance. You can alter its value during a session using an ALTER SESSION SET NLS_ISO_CURRENCY statement.

Typical ISO currency symbols are shown in [Table 3-10](#).

Table 3–10 ISO Currency Examples

Country	Example
Estonia	1 234 567,89 EEK
Germany	1.234.567,89 DEM
Japan	1,234,567.89 JPY
UK	1,234,567.89 GBP
US	1,234,567.89 USD

NLS_DUAL_CURRENCY

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Default dual currency symbol for a particular territory
Range of values:	Any valid name

You can use this parameter to override the default dual currency symbol defined in the territory. When starting a new session without setting `NLS_DUAL_CURRENCY`, you will use the default dual currency symbol defined in the territory of your current language environment. When you set `NLS_DUAL_CURRENCY`, you will start up a session with its value as the dual currency symbol.

Simon-check these

`NLS_DUAL_CURRENCY` was introduced to help support the Euro. The following [Table 3–11](#) lists the character sets that support the Euro symbol:

Table 3–11 Character Sets that Support the Euro Symbol

Name	Description	Euro Code Value
D8EBCDIC1141	EBCDIC Code Page 1141 8-bit Austrian German	0x9F
DK8EBCDIC1142	EBCDIC Code Page 1142 8-bit Danish	0x5A
S8EBCDIC1143	EBCDIC Code Page 1143 8-bit Swedish	0x5A
I8EBCDIC1144	EBCDIC Code Page 1144 8-bit Italian	0x9F
F8EBCDIC1147	EBCDIC Code Page 1147 8-bit French	0x9F
WE8PC858	IBM-PC Code Page 858 8-bit West European	0xDF
WE8ISO8859P15	ISO 8859-15 West European	0xA4

Table 3–11 Character Sets that Support the Euro Symbol (Cont.)

EE8MSWIN1250	MS Windows Code Page 1250 8-bit East European	0x80
CL8MSWIN1251	MS Windows Code Page 1251 8-bit Latin/Cyrillic	0x88
WE8MSWIN1252	MS Windows Code Page 1252 8-bit West European	0x80
EL8MSWIN1253	MS Windows Code Page 1253 8-bit Latin/Greek	0x80
WE8EBCDIC1047E	Latin 1/Open Systems 1047	
WE8EBCDIC1140	EBCDIC Code Page 1140 8-bit West European	0x9F
WE8EBCDIC1140C	EBCDIC Code Page 1140 Client 8-bit West European	0x9F
WE8EBCDIC1145	EBCDIC Code Page 1145 8-bit West European	0x9F
WE8EBCDIC1146	EBCDIC Code Page 1146 8-bit West European	0x9F
WE8EBCDIC1148	EBCDIC Code Page 1148 8-bit West European	0x9F
WE8EBCDIC1148C	EBCDIC Code Page 1148 Client 8-bit West European	0x9F
EL8ISO8859P7	ISO 8859-7 Latin/Greek	0xA4
IW8MSWIN1255	MS Windows Code Page 1255 8-bit Latin/Hebrew	0x80
AR8MSWIN1256	MS Windows Code Page 1256 8-Bit Latin/Arabic	0x80
TR8MSWIN1254	MS Windows Code Page 1254 8-bit Turkish	0x80
BLT8MSWIN1257	MS Windows Code Page 1257 Baltic	0x80
VN8MSWIN1258	MS Windows Code Page 1258 8-bit Vietnamese	0x80
TH8TISASCII	Thai Industrial 620-2533 - ASCII 8-bit	0x80
AL32UTF8	Unicode 3.0 UTF-8 Universal character set	E282AC
UTF8	Unicode 3.0 UTF-8 Universal character set	E282AC
AL16UTF16	Unicode 3.0 UTF-16 Universal character set	20AC
UTFE	UTF-EBCDIC encoding of Unicode 3.0	CA4653
ZHT16HKSCS	MS Windows Code Page 950 with Hong Kong Supplementary Character Set	
ZHS32GB18030	GB18030-2000	
WE8BS2000E	Siemens EBCDIC.DF.04 8-bit West European	

NLS_MONETARY_CHARACTERS

Parameter type: String
Parameter scope: Environment Variable

Default value:	Derived from NLS_TERRITORY
Range of values:	Any valid name

NLS_MONETARY_CHARACTERS specifies the characters that indicate monetary units, such as the dollar sign (\$) for U.S. dollars, and the cent symbol (¢) for cents.

The two characters specified must be single-byte and cannot be the same as each other. They also cannot be any numeric character or any of the following characters: plus (+), hyphen (-), less than sign (<), greater than sign (>).

NLS_CREDIT

Parameter type:	String
Parameter scope:	Environment Variable
Default value:	Derived from NLS_TERRITORY
Range of values:	Any string, maximum of 9 bytes (not including null)

NLS_CREDIT sets the symbol that displays a credit in financial reports. The default value of this parameter is determined by NLS_TERRITORY.

This parameter can be specified only in the client environment. It can be retrieved through the OCIGetNlsInfo function.

NLS_DEBIT

Parameter type:	String
Parameter scope:	Environment Variable
Default value:	Derived from NLS_TERRITORY
Range of values:	Any string, maximum of 9 bytes (not including null)

NLS_DEBIT sets the symbol that displays a debit in financial reports. The default value of this parameter is determined by NLS_TERRITORY.

This parameter can be specified only in the client environment. It can be retrieved through the OCIGetNlsInfo function.

Linguistic Sorting Parameters

Oracle allows you to choose how data is sorted through the use of linguistic parameters.

Oracle provides many different types of sorts, but achieving a linguistically correct sort frequently harms performance. This is a trade-off the database administrator needs to make on a case-by-case basis. A typical case would be when sorting Spanish. In traditional Spanish, `ch` and `ll` are distinct characters, which means that the correct order would be: `cerveza`, `colorado`, `cheremoya`, `lago`, `luna`, `llama`.

See Also: [Chapter 4, "Linguistic Sorting"](#)

NLS_SORT

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Default character sort sequence for a particular language
Range of values:	BINARY or any valid linguistic definition name

This parameter specifies the type of sort for character data, overriding that defined implicitly by `NLS_LANGUAGE`.

The syntax of `NLS_SORT` is:

```
NLS_SORT = { BINARY | name }
```

`BINARY` specifies a binary sort and *name* specifies a particular linguistic sort sequence. For example, to specify the linguistic sort sequence called German, the parameter should be set as follows:

```
NLS_SORT = German
```

The name given to a linguistic sort sequence has no direct connection to language names. Usually, however, each supported language has an appropriate linguistic sort sequence defined that uses the same name.

Note: When the `NLS_SORT` parameter is set to `BINARY`, the optimizer can, in some cases, satisfy the `ORDER BY` clause without doing a sort (by choosing an index scan).

When `NLS_SORT` is set to a linguistic sort, a sort is always needed to satisfy the `ORDER BY` clause if the linguistic index does not exist for the linguistic sort order specified by `NLS_SORT`. If the linguistic index exists for the linguistic sort order specified by `NLS_SORT`, the optimizer can, in some cases, satisfy the `ORDER BY` clause without doing a sort (by choosing an index scan).

You can alter the default value of `NLS_SORT` by changing its value in the initialization parameter file and then restarting the instance. You can alter its value during a session using an `ALTER SESSION SET NLS_SORT` statement.

A complete list of linguistic definitions is provided in [Table A-10, "Monolingual Linguistic Sorts"](#).

NLS_COMP

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable and <code>ALTER SESSION</code>
Default value:	Binary
Range of values:	<code>BINARY</code> or <code>ANSI</code>

You can use this parameter to avoid the cumbersome process of using `NLS_SORT` in SQL statements. Normally, comparison in the `WHERE` clause and in PL/SQL blocks is binary. To use linguistic comparison, the `NLSSORT` function must be used. Sometimes this can be tedious, especially when the linguistic sort needed has already been specified in the `NLS_SORT` session parameter. You can use `NLS_COMP` in such cases to indicate that the comparisons must be linguistic according to the `NLS_SORT` session parameter. This is done by altering the session:

```
ALTER SESSION SET NLS_COMP = ANSI;
```

To specify that comparison in the `WHERE` clause is always binary, issue the following statement:

```
ALTER SESSION SET NLS_COMP = BINARY;
```

As a final note, when `NLS_COMP` is set to `ANSI`, a linguistic index improves the performance of the linguistic comparison.

To enable a linguistic index, use the syntax:

```
CREATE INDEX i ON t(NLSSORT(col, 'NLS_SORT=FRENCH'));
```

NLS_LIST_SEPARATOR

Parameter type:	String
Parameter scope:	Environment Variable
Default value:	Derived from <code>NLS_TERRITORY</code>
Range of values:	Any valid character

`NLS_LIST_SEPARATOR` specifies the character to use to separate values in a list of values.

The character specified must be single-byte and cannot be the same as either the numeric or monetary decimal character, any numeric character, or any of the following characters: plus (+), hyphen (-), less than sign (<), greater than sign (>), period (.).

Character Set Parameter

You can specify the character set used for the client.

NLS_NCHAR_CONV_EXCP

Parameter type:	String
Parameter scope:	Environment Variable, ALTER SYSTEM, ALTER SESSION
Default value:	FALSE
Range of values:	TRUE, FALSE

`NLS_NCHAR_CONV_EXCP` determines whether data loss during an implicit or explicit character type conversion will report an error.

Linguistic Sorting

This chapter explains how characters are sorted in an Oracle environment. Its sections are:

- [Overview of Oracle's Sorting Capabilities](#)
- [Using Binary Sorts](#)
- [Using Linguistic Sorts](#)
- [Using Linguistic Indexes](#)

Overview of Oracle's Sorting Capabilities

Different languages have different sort orders. What's more, different cultures or countries using the same alphabets may sort words differently. For example, in Danish, the letter Æ is after Z, while ŷ and Ÿ are considered to be variants of the same letter. Sort order can be case sensitive or insensitive and can ignore accents or not. It can also be either phonetic or based on the appearance of the character, such as ordering by the number of strokes or by radicals for East Asian ideographs. Another common sorting issue is when letters are combined. For example, in traditional Spanish, *ch* is a distinct character, which means that the correct order is: *cerveza*, *colorado*, *cheremoya*, and so on. This means that the letter *c* cannot be sorted until checking to see if the next letter is an *h*.

Oracle provides several different types of sort, and can achieve a linguistically correct sort as well as the new multilingual ISO standard (14651), which is designed to handle many languages at the same time.

Using Binary Sorts

Conventionally, when character data is stored, the sort sequence is based on the numeric values of the characters defined by the character encoding scheme. This is called a **binary sort**. Binary sorts are the fastest type of sort, and produce reasonable results for the English alphabet because the ASCII and EBCDIC standards define the letters A to Z in ascending numeric value. Note, however, that in the ASCII standard, all uppercase letters appear before any lowercase letters. In the EBCDIC standard, the opposite is true: all lowercase letters appear before any uppercase letters.

When characters used in other languages are present, a binary sort generally does not produce reasonable results. For example, an ascending `ORDER BY` query would return the character strings ABC, ABZ, BCD, ÄBC, in the sequence, when the Ä has a higher numeric value than B in the character encoding scheme. For languages using Chinese characters, a binary sort is not usually linguistically meaningful.

Using Linguistic Sorts

To produce a sort sequence that matches the alphabetic sequence of characters, another sort technique must be used that sorts characters independently of their numeric values in the character encoding scheme. This technique is called a **linguistic sort**. A linguistic sort operates by replacing characters with numeric values that reflect each character's proper linguistic order.

Oracle offers two kinds of linguistic sorts:

- [Monolingual Linguistic Sorts](#), commonly used for European languages
- [Multilingual Linguistic Sorts](#), commonly used for Asian languages

Monolingual Linguistic Sorts

Oracle makes two passes when comparing strings in monolingual sorts. The first pass is to compare the major value of entire string from the major table and the second pass is to compare the minor value from the minor table. Each major table entry contains the Unicode codepoint and major value. Usually, letters with the same appearance will have the same major value. Oracle defines letters with diacritic and case differences for the same major value but different minor values. [Table 4–1](#) illustrates sample values for sorting a, A, ä, and Ä.

Table 4–1 Sample Glyphs and Their Major and Minor Codepoint Values

Glyph	Major Value	Minor Value
a	15	5
A	15	10
ä	15	15
Ä	15	20

Multilingual Linguistic Sorts

Oracle9i provides multilingual linguistic sorts so that you can sort more than one language as part of one sort. This is useful for certain regions or languages that have complex sorting rules or global multilingual databases. Additionally, Oracle9i still supports all the sort orders defined by previous releases.

For Asian language data or multilingual data, Oracle provides a sorting mechanism based on an ISO standard (ISO14651) and the Unicode 3.0 standard. Multilingual linguistic sorts also work for Asian language sorts ordered by the number of strokes, PinYin, or radicals. In addition, they can handle canonical equivalence and surrogate codepoint pairs. You can define up to 1.1 million codepoints in one sort.

For example, in Oracle9i, a French sort is supported, but the new multilingual linguistic sort for French can also be applied by changing the sort order from French to French_M, where _M represents the new ISO standard (ISO 14651) for multilingual sorting. By doing so, the sorting order will be based on the GENERIC_M sorting order (ISO standard) and will be able to sort at the secondary level from

right to left. Oracle Corporation recommends using a multilingual linguistic sort if the tables contain multilingual data. If the tables contain only pure French, for memory usage concern, a French monolingual sort may get better performance. You have to make a trade-off between extensibility and performance.

To use a multilingual linguistic sort, you can specify the default sort order by setting the environment variable `NLS_SORT` or using the `NLSSORT` function.

```
% setenv NLS_SORT='French_M'
```

or

```
NLSSORT(' ', 'NLS_SORT=French_M')
```

You can create new linguistic indexes based on multilingual linguistic sorts just as you did for monolingual linguistic sorts. However, if you do not want to change every index hint of your SQL statements, but you do need to use those new multilinguistic sorts. Here are the steps:

```
DROP INDEX index_table1;  
CREATE INDEX index_table1 ON table1(col, 'NLS_SORT=French_M');  
COMMIT;
```

Multilingual Sorting Levels

Oracle evaluates multilingual sorts at three levels of precision:

- [Primary Level Sorts](#)
- [Secondary Level Sorts](#)
- [Tertiary Level Sorts](#)

Primary Level Sorts A primary level sort distinguishes between **base characters**, such as the difference between characters a and b. It is up to individual locales to define if a is before b, b is before a, or they are equal. The binary representation of the characters is completely irrelevant. If a character is an ignorable character, it is assigned a primary level weight (or order) of zero, which means it is ignored at the primary level. Ignorable characters on all other levels are also defined by the use of weight zero. At the primary level, the following words are not distinguished between uppercase and lowercase words and can appear in different orders.

```
Bat  
bat  
BAT  
BET  
Bet
```

bet

Secondary Level Sorts A secondary level sort distinguishes between base characters (the primary level sort), plus it distinguishes the different diacritical marks on a given base character. For example, the character Å differs from the character A only because it has a diacritical mark. Thus, Å and A differ on the secondary level but they are the same on the primary level because they are derived from the same base character A.

```
SELECT words FROM rdictionary; --with a secondary level sort
words
-----
resume
résumé
Résumé
Resumes
resumes
résumés
```

Tertiary Level Sorts A tertiary level sort distinguishes between base characters (primary level sort), diacritical marks (secondary level sort), and the different cases of characters. It can also include difference of special characters such as +, -, and *. For example, characters a and A are different on the tertiary level and equal on the primary and secondary levels because they only have a case difference. Another example is that characters ä and Å are equal on the primary level and different on secondary and tertiary levels. The final example is that the primary order for the dash character - is 0. That is, it is ignored on the primary and secondary levels. If a dash is compared with another character whose primary level order is nonzero, for example, the character u, then no result for the primary level is available because u is not compared with anything. In this case, Oracle finds a difference between - and u, but only at the tertiary level. For example:

```
SELECT words FROM rdictionary; --with a tertiary level sort
words
-----
resume
Resume
résumé
Résumé
resumes
résumés
Resumes
Résumés
```

General Linguistic Sorting Information

You should consider the following issues when sorting:

- [Ignorable Characters](#)
- [Contracting Characters](#)
- [Expanding Characters](#)
- [Context-Sensitive Characters](#)
- [Canonical Equivalence](#)
- [Surrogate Characters](#)
- [Reverse Secondary Sorting](#)
- [Character Rearrangement for Thai/Lao Characters](#)

Ignorable Characters

When sorting, some characters can or should be ignored. For example, a dash in multi-lingual could be treated the same for sorting purposes as multilingual. These characters are called **ignorable characters**. There are two kinds of ignorable characters: accents and punctuation.

Examples of ignorable accent characters:

- rôle can be treated as role
- naïve can be treated as naive

Examples of ignorable punctuation characters:

- multi-lingual can be treated as multilingual
- e-mail can be treated as email

Contracting Characters

Sorting elements usually consist of a single character, but, in some locales, two or more characters in a character string must be considered as a single sorting element during sorting. For example, in Spanish, the string `ch` is composed of two characters. These characters are called **contracting characters** or **group characters** in multilingual linguistic sorting and **special combination letters** in monolingual linguistic sorting. The important difference is that a composed character can be displayed as a single character on a terminal (if desired), while a contracting character is only used for sorting, and its component characters must be rendered separately. Note that a contracting character is not a Unicode-composed character.

Expanding Characters

In some locales, one character must be sorted as if it was a character string. An example is the German character ß (sharp s). It is sorted exactly the same as the string ss. Another example is that ö sorts as if it were oe, after od and before of. These characters are known as **expanding characters** in multilingual linguistic sorting and **special letters** in monolingual linguistic sorting. Just as with contracting characters, the replacement string for an expanding character is only meaningful for sorting.

Context-Sensitive Characters

In Japanese, a prolonged sound mark (resembles an em dash —) represents a length mark that lengthens the vowel of the preceding character. Depending on the vowel, the result will sort in a different order. This is called context-sensitive sorting. For example, after the character ka, the — length mark indicates a long a and is treated the same as a, while after the character ki, the — length mark indicates a long i and is treated the same as i. Transliterating this to Latin characters, a sort might look like this:

```
kaa  -- kaa and ka- are the same
ka-
kai  -- kai follows ka- because i is after a
kia
kii  -- kii and ki- are the same
ki-
```

Canonical Equivalence

One Unicode code point may be equivalent to a sequence of base character code points plus combining characters (accent marks) code points, regardless of locales in use. This is called the Unicode canonical equivalence. For example, â equals its base letter a and a combining character diaeresis. A linguistic flag, `Canonical_equivalence=TRUE`, which you can set in the definition file, indicates that all canonical equivalence rules defined in Unicode 3.0 need to be applied. You can change this flag to `FALSE` to speed up the comparison and ordering functions if all the data is in its composed form.

Surrogate Characters

You can extend UTF-16 and UTF-8 to encode more than 1 million characters. These extended characters are called surrogate pairs. Multilingual linguistic sorts can support up to one million surrogate pairs. However, surrogate characters cannot be

defined as contracting characters, expanding characters, or context-sensitive characters.

Reverse Secondary Sorting

In French, the sorting of strings with accented characters compares base characters from left to right, but compares accented characters from right to left. For example, by default, an accented character is placed after its unaccented variant. Then the two strings `Édit` and `Edít` are in proper French order. They are equal on the primary level, and the secondary order is determined by examining accented characters from right to left. Individual locales can request that the accented characters and related diacritical marks be sorted with the right-to-left rule. This is specified in a locale specification file by using a linguistic flag.

See Also: [Chapter 11, "Oracle Locale Builder Utility"](#) for more information about customizing a sort

Character Rearrangement for Thai/Lao Characters

In Thai and Lao, some characters must first be swapped with their following character before sorting. Normally, these types of character are symbols representing vowel sounds, and the next character will be a consonant. Consonants and vowels must be swapped before sorting. A linguistic flag is used to enable you to specify all the characters to be swapped.

Base Letters

Base letters are defined in a base letter table, which maps each letter to its base letter. For example, `a`, `À`, `ä`, and `Ä` all map to `a`, which is the **base letter**. This concept is particularly relevant for working with Oracle9i Text.

See Also: *Oracle Text Reference*

Special Letters

Special letters is the term used in monolingual sorts. They are called **expanding characters** in multilingual sorts.

See Also: ["Expanding Characters"](#) on page 4-7

Special Combination Letters

Special combination letters is the term used in monolingual sorts. They are called **contracting letters** in multilingual sorts.

See Also: ["Contracting Characters"](#) on page 4-6

Special Uppercase Letters

One lowercase letter may map to multiple uppercase letters. For example, in traditional German, the uppercase letters for ß are SS.

Special cases like these are also handled when converting uppercase characters to lowercase, and vice versa. Such case-conversion issues are handled by the `NLS_UPPER`, `NLS_LOWER`, and `NLS_INITCAP` functions, according to the conventions established by the linguistic sort sequence. (The standard functions `UPPER`, `LOWER`, and `INITCAP` cannot handle these special cases.)

Special Lowercase Letters

Oracle supports special lowercase letters, so one letter may map to multiple base letters. An example is the Turkish uppercase İ becoming a small, dotless i.

Using Linguistic Indexes

Linguistic sorting is language-specific and requires more data processing than binary sorting. Binary sorting ASCII is accurate and fast because it is in order. When data of multiple languages is stored in the database, you may want your applications to collate a result set returned from a `SELECT` statement using the `ORDER BY` clause with different collating sequences based upon the language being used. You can accomplish this without sacrificing performance by using linguistic indexes, a feature introduced in Oracle8i. While a linguistic index for a column slows down inserts and updates, it greatly improves the performance of linguistic sorting with the `ORDER BY` clause.

You can create a function-based index that uses languages other than English. The index does not change the linguistic sort order determined by `NLS_SORT`. The index simply improves the performance. An example is:

```
CREATE INDEX nls_index ON my_table (NLSSORT(name, 'NLS_SORT = German'));
```

So

```
SELECT * FROM my_table WHERE NLSSORT(name) IS NOT NULL
ORDER BY name;
```

returns the result much faster than without an index.

See Also: *Oracle9i Database Concepts* for more information about function-based indexes

Linguistic Indexes for Multiple Languages

There are three ways to build linguistic indexes for data in multiple languages:

- Build a linguistic index for each language that the application needs to support. This approach offers simplicity but requires more disk space. For each index, the rows in the language other than the one on which the index is built are collated together at the end of the sequence. The following example builds linguistic indexes for French and German.

```
CREATE INDEX french_index ON emp (NLSSORT(emp_name, 'NLS_SORT=FRENCH'));
CREATE INDEX german_index ON emp (NLSSORT(emp_name, 'NLS_SORT=GERMAN'));
```

Which index to use is based on the NLS_SORT session parameters or the arguments of the NLSSORT function you specified in the ORDER BY clause. For example, if the session variable NLS_SORT is set to FRENCH, you can use french_index and when it is set to GERMAN, you can use german_index.

- Build a single linguistic index for all languages. This can be accomplished by including a language column (LANG_COL in the example below) that contains NLS_LANGUAGE values for the corresponding column on which the index is built as a parameter to the NLSSORT function. The following example builds a single linguistic index for multiple languages. With this index, the rows with the same values for NLS_LANGUAGE are collated together.

```
CREATE INDEX i ON t (NLSSORT(col, 'NLS_SORT=' || LANG_COL));
```

Which index to use is based on the argument of the NLSSORT function you specified in the ORDER BY clause.

- Build a single linguistic index for all languages using one of the sorting sequences such as GENERIC_M or FRENCH_M. These indexes collate characters according to the rules defined in ISO 14651.

```
CREATE INDEX i on t (NLSSORT(col, 'NLS_SORT=GENERIC_M'));
```

See Also: [Chapter 4, "Linguistic Sorting"](#) for more information about the different Unicode sorting sequences

Requirements for Linguistic Indexes

If you want to use a single linguistic index or multiple linguistic indexes, some requirements must be met for the linguistic index. The first requirement is that the `QUERY_REWRITE_ENABLED` initialization parameter must be set to `TRUE`. This is not a specific requirement for linguistic indexes, but for all function-based indexes. Here is an example of setting `QUERY_REWRITE_ENABLED`:

```
ALTER SESSION SET QUERY_REWRITE_ENABLED=TRUE;
```

The second requirement, which is specific to linguistic indexes, is that `NLS_COMP` needs to be set to `ANSI`. There are various ways to set `NLS_COMP`. For example:

```
ALTER SESSION SET NLS_COMP = ANSI;
```

The third requirement is that `NLS_SORT` needs to indicate the linguistic definition you want to use for the linguistic sort. If you want a French linguistic sort order, `NLS_SORT` needs to be set to `FRENCH`. If you want a German linguistic sort order, `NLS_SORT` needs to be set to `GERMAN`.

There are various ways to set `NLS_SORT`. Although the following example uses an `ALTER SESSION` statement, it is probably better for you to set `NLS_SORT` as a client environment variable so that you can use the same SQL statements for all languages and different linguistic indexes can be used, based on `NLS_SORT` being set in the client environment. The following is an example of setting `NLS_SORT`:

```
ALTER SESSION SET NLS_SORT='FRENCH';
```

The fourth requirement is that you need to use the cost-based optimizer with the optimizer mode set to `FIRST_ROWS`, because linguistic indexes are not recognized by the rule-based optimizer. The following is an example of setting the optimizer mode:

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS;
```

The last thing is that you need to set `WHERE NLSSORT(column_name) IS NOT NULL` when you want to use `ORDER BY column_name`, where `column_name` is the column with the linguistic index. This is necessary only when you use an `ORDER BY` clause.

The following example shows how to set up a French linguistic index. For NLS_SORT, you may want to set it in the client environment variable instead of with the ALTER SESSION statement.

```
ALTER SESSION SET QUERY_REWRITE_ENABLED=TRUE;
ALTER SESSION SET NLS_COMP = ANSI;
ALTER SESSION SET NLS_SORT='FRENCH';
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS;
CREATE TABLE test(col VARCHAR(20) NOT NULL);
CREATE INDEX test_idx ON test(NLSSORT(col, 'NLS_SORT=FRENCH'));
SELECT * FROM test WHERE NLSSORT(col) IS NOT NULL ORDER BY col;
SELECT * FROM test WHERE col > 'JJJ';
```

See Also: *Oracle9i Database Concepts* for more information about function-based indexes

Case-Insensitive Searching

You can create a function-based index that improves the performance of case-insensitive searches. For example:

```
CREATE INDEX case_insensitive_ind ON my_table(NLS_UPPER(empname));
SELECT * FROM my_table WHERE NLS_UPPER(empname) = 'KARL';
```

Customizing Linguistic Sorts

You can customize sorting with the Locale Builder Utility.

See Also: [Chapter 11, "Oracle Locale Builder Utility"](#)

Supporting Multilingual Databases with Unicode

This chapter illustrates how to use Unicode in an Oracle database environment. It includes the following topics:

- [Overview of Unicode](#)
- [Implementing a Unicode Solution in the Database](#)
- [Unicode Case Studies](#)
- [Migrating Data to Unicode](#)
- [Designing Database Schemas to Support Multiple Languages](#)

Overview of Unicode

Dealing with many different languages in the same application or database has been complicated and difficult for a long time. To overcome the limitations of existing character encodings, several organizations began working on the creation of a global character set in the late 1980s. The need for this became even greater with the development of the World Wide Web in the mid-1990s. The Internet has changed how we do business today, with an emphasis on the global market that has made a universal character set a major requirement. This global character set needs to contain all major living scripts, support legacy data and implementations, and be simple enough that a single implementation of a product is sufficient for worldwide use. This character set should also support multilingual users and organizations, conform to international standards, and enable world-wide interchange of data. This global character set exists, is in wide use, and is called Unicode.

What is Unicode?

Unicode is a universal encoded character set that allows you to store information from any language using a single character set. Unicode provides a unique code value for every character, regardless of the platform, program, or language.

The Unicode standard has been adopted by many software and hardware vendors. Many operating systems and browsers now support Unicode. Unicode is required by modern standards such as XML, Java, JavaScript, LDAP, CORBA 3.0, and WML. It is also synchronized with the ISO/IEC 10646 standard.

Oracle started supporting Unicode as a database character set in Oracle7. In Oracle9i, Unicode support has been expanded so that you can find the right solution for your globalization needs. Oracle9i supports Unicode 3.0, the third version of the Unicode standard.

See Also: <http://www.unicode.org> for more information about the Unicode standard

Unicode Encoding

There are two different ways to encode Unicode 3.0 characters:

- [UTF-16 Encoding](#)
- [UTF-8 Encoding](#)

UTF-16 Encoding

UTF-16 encoding is the 16-bit encoding of Unicode in which the character codes 0x0000 through 0x007F contain the ASCII characters. One Unicode character can be 2 bytes or 4 bytes in this encoding. Characters from both European and most Asian scripts are represented in 2 bytes. Surrogate pairs (described below) are represented in 4 bytes.

UTF-8 Encoding

UTF-8 is the 8-bit encoding of Unicode. It is a variable-width encoding in which the character codes 0x00 through 0x7F have the same meaning as ASCII. One Unicode character can be 1 byte, 2 bytes, 3 bytes or 4 bytes in this encoding. Characters from the European scripts are represented in either 1 or 2 bytes, while characters from most Asian scripts are represented in 3 bytes. Surrogate pairs (described below) are represented in 4 bytes.

Note: Converting from UTF-8 to UTF-16 is a simple bit-wise operation that is defined in the Unicode standard

Surrogate Pairs

You can extend Unicode to encode more than 1 million characters. These extended characters are called surrogate pairs. Surrogate pairs are designed to allow representation of characters in Unicode 3.0 and future extensions of the Unicode standard. Surrogate pairs require 4 bytes in UTF-8 and UTF-16 encoding.

Sample UTF-16 and UTF-8 Encodings

Figure 5–1 shows some characters and their character codes in UTF-16 and UTF-8 encoding. The last character is a treble clef (a music symbol), a surrogate pair that has been added to the Unicode 3.0 standard.

Figure 5–1 Sample UTF-16 and UTF-8 Encodings

Char	UTF-16	UTF-8
A	0041	41
c	0063	63
ö	00F6	C3 86
𐄌	4E9C	E4 BA 9C
🎵	D834 DD1E	F0 9D 84 9E

Implementing a Unicode Solution in the Database

You can store Unicode characters in an Oracle9i database in two ways. The following sections explain how to use the two Unicode solutions and how to choose between them:

- [Enabling Multilingual Support with Unicode Databases](#)
- [Enabling Multilingual Support with Unicode Datatypes](#)

You can create a Unicode database that allows you to store UTF-8 encoded characters as SQL CHAR datatypes (CHAR, VARCHAR2, CLOB, and LONG). If you prefer to implement Unicode support incrementally, you can store Unicode data in either the UTF-16 or UTF-8 encoding form in SQL NCHAR datatypes (NCHAR, NVARCHAR2, and NCLOB). The SQL NCHAR datatypes are called Unicode datatypes because they are used for storing Unicode data only.

Note: You can combine a Unicode database solution with a Unicode datatype solution.

Enabling Multilingual Support with Unicode Databases

The Oracle9i database has the concept of a database character set, which specifies the encoding to be used in the SQL CHAR datatypes as well as the metadata such as table names, column names, and SQL statements. A Unicode database is a database with UTF-8 as the database character set. There are three Oracle character sets that implement the UTF-8 encoding. The first two are designed for ASCII-based platforms while the third one should be used on EBCDIC platforms.

- **AL32UTF8**

The AL32UTF8 character set encodes characters in one to three bytes. Surrogate pairs require four bytes. It is for ASCII-based platforms.

- **UTF8**

The UTF8 character set encodes characters in one to three bytes. Surrogate pairs require six bytes. It is for ASCII-based platforms.

- **UTFE**

The UTFE character set should be used as the database character set on EBCDIC platforms to support the UTF-8 encoding.

Note: The UTF8 character set is used throughout this chapter, but all technical details apply to the other two character sets unless otherwise specified

Example 5–1 Creating a Database with a Unicode Character Set

To create a database with the UTF8 character set, use the CREATE DATABASE statement. For example:

```
CREATE DATABASE myunicodedatabase
  CONTROL FILE REUSE
  LOGFILE '/u01/oracle/utfdb/redo01.log' SIZE 1M REUSE
        '/u01/oracle/utfdb/redo02.log' SIZE 1M REUSE
  DATAFILE '/u01/oracle/utfdb/system01.dbf' SIZE 10M REUSE
    AUTOEXTENT ON
    NEXT 10M MAXSIZE 200M
  CHARACTER SET UTF8
```

Note: The database character set needs to be specified when you create the database

Enabling Multilingual Support with Unicode Datatypes

An alternative to storing Unicode data in the database is to use the SQL NCHAR datatypes. You can store Unicode characters into columns of these datatypes regardless of how the database character set has been defined. The NCHAR datatype has been redefined in Oracle9i to be a Unicode datatype exclusively. In other words, it stores data in the Unicode encoding only. You can use the SQL NCHAR datatypes

in the same way you use the SQL CHAR datatypes. You can create a table using the NVARCHAR2 and NCHAR datatypes as follows:

```
CREATE TABLE product
(id          NUMBER
ename       NCHAR(100)
description NVARCHAR2(1000))
```

The encoding used in the SQL NCHAR datatypes is specified as the national character set of the database. You can specify one of the following two Oracle character sets as the national character set:

- **AL16UTF16**

This is the default character set for SQL NCHAR datatypes. The character set encodes Unicode data in the UTF-16 encoding.

- **UTF8**

When UTF8 is specified for SQL NCHAR datatypes, the data stored in the SQL datatypes is in UTF-8 encoding.

By default, data is stored in the UTF-16 encoding in the SQL NCHAR datatypes, and the length specified in the NCHAR and NVARCHAR2 columns is always in the number of characters instead of the number of bytes.

You can specify the national character set for the SQL NCHAR datatypes when you create a database using the CREATE DATABASE statement. The following command creates a database with WE8ISO8859P1 as the database character set and AL16UTF16 as the national character set.

Example 5–2 Creating a Database with a National Character Set

```
CREATE DATABASE myunicodedatabase
CONTROL FILE REUSE
LOGFILE '/u01/oracle/utfdb/redo01.log' SIZE 1M REUSE
        '/u01/oracle/utfdb/redo02.log' SIZE 1M REUSE
DATAFILE '/u01/oracle/utfdb/system01.dbf' SIZE 10M REUSE
        AUTOEXTENT ON
        NEXT 10M MAXSIZE 200M
CHARACTER SET WE8ISO8859P1
NATIONAL CHARACTER SET AL16UTF16
```

This example also illustrates the fact that you can store Unicode data in a non-Unicode database by using SQL NCHAR datatypes.

See Also: ["Migrating to Use the NCHAR Datatypes"](#) on page 5-16 for more information about migrating data to the NCHAR datatype

How to Choose Between a Unicode Database and a Unicode Datatype Solution

In order to choose the right Unicode solution for your database, you need to consider the following:

- **Programming environment**
What are the main programming languages used in your applications? How do they support Unicode?
- **Ease of migration**
How easily can your data and applications be migrated to take advantage of the Unicode solution?
- **Performance**
How much performance overhead are you willing to accept in order to use Unicode in the database?
- **Type of data**
Is your data mostly Asian or European? Do you need to store multilingual documents into LOB columns?
- **Type of applications**
What type of applications are you implementing: a packaged application or a customized end-user application?

This section describes some general guidelines for choosing a Unicode database or a Unicode datatype solution. The final decision largely depends on your exact environment and requirements.

When Should You Use a Unicode Database?

You should use a Unicode database when:

1. **You need easy code migration for Java or PL/SQL.**
If your existing application is mainly written in Java and PL/SQL and your main concern is to minimize the code change required to support multiple languages, you may want to use a Unicode database solution. As long as the datatypes used to store your data remain as the SQL CHAR datatypes, the Java and PL/SQL accessing these columns do not need to change.

2. You need easy data migration from ASCII.

If the database character set of your existing database is US7ASCII, you may want to choose a Unicode database solution because you can migrate your database using a single `ALTER DATABASE` statement. No data conversion is required because ASCII is a subset of UTF-8.

3. You have evenly distributed multilingual data.

If the multilingual data will be evenly distributed in your existing schema tables and you are not sure which ones will contain multilingual data, then you should use the Unicode database solution because it does not require you to identify which columns store what data.

4. Your SQL statements and PL/SQL code contain Unicode data.

If you need to write SQL statements and PL/SQL code in Unicode, you must use the Unicode database solution. SQL statements and PL/SQL code are converted into the database character set before being processed. If your SQL statements and PL/SQL code contain characters that cannot be converted to your database character set, those characters will be lost. A common place where you would use Unicode data in a SQL statement is in a string literal.

5. You want to store multilingual documents as BLOBs and use Oracle Text for content searching.

You must use a Unicode database in this case. The BLOB data is converted to the database character set before being indexed by Oracle Text. If your database character set is not UTF8, then data will be lost when the documents contain characters that cannot be converted to the database character set.

After you decide to use a Unicode database solution, you need to decide which UTF-8 character set is best for your database. See [Table 5–1](#) for advantages and disadvantages of Unicode database solutions.

When Should You Use Unicode Datatypes?

You should use Unicode datatypes when:

1. You want to add multilingual support incrementally.

If you want to add Unicode support to your existing database without migrating the character set, you should consider using Unicode datatypes to store Unicode. You can add columns of the SQL `NCHAR` datatypes to existing tables or new tables to support multiple languages incrementally.

2. You want to build a packaged application.

If you are building a packaged application that will be sold to customers, you may want to build the application using SQL `NCHAR` datatypes. This is because the SQL `NCHAR` datatype is a reliable Unicode datatype in which the data is always stored in Unicode, and the length of the data is always specified in UTF-16 code units. As a result, you need only test the application once, and your application will run on customer databases with any database character set.

3. You want better performance with single-byte database character sets.

If performance is your biggest concern, you should consider using a single-byte database character set and storing Unicode data in the SQL `NCHAR` datatypes. Databases using a multibyte database character set such as UTF8 have a performance overhead.

4. You require UTF-16 support in Windows clients.

If your applications are written in Visual C/C++ or Visual Basic running on Windows, you may want to use the SQL `NCHAR` datatypes because you can store UTF-16 data in these datatypes in the same way that you store it in the `wchar_t` buffer in Visual C/C++ and `string` buffer in Visual Basic. You can avoid buffer overflow in your client applications because the length of the `wchar_t` and `string` datatypes match the length of the SQL `NCHAR` datatypes in the database.

Once you decide to use a Unicode datatype solution, you need to decide which character set to choose as your national character set. See [Table 5-2](#) for advantages and disadvantages of Unicode datatype solutions.

Note: You can use a Unicode database in conjunction with Unicode datatypes

Comparison of Unicode Solutions

Oracle9i provides two solutions to store Unicode characters in the database: a Unicode database solution and a Unicode datatype solution. After you select the Unicode database solution, the Unicode datatype solution or the combination of both, you need to determine the character set to be used in the Unicode database and/or the Unicode datatype. Different Unicode character sets have different advantages and disadvantages. [Table 5-1](#) and [Table 5-2](#) illustrate advantages and disadvantages for different Unicode solutions:

Table 5–1 Unicode Database Solutions

Database Character Set	Advantages	Disadvantages
AL32UTF8	<ol style="list-style-type: none"> 1. Surrogate pair Unicode characters are stored in the standard 4 bytes representation, and there is no data conversion upon retrieval and insertion of those surrogate characters. Also, the storage for those characters requires less disk space than that of the same characters encoded in UTF8. 	<ol style="list-style-type: none"> 1. You cannot specify the length of SQL CHAR types in the number of characters (Unicode code points) for surrogate characters. For example, surrogate characters are treated as one code point rather than the standard of two code points. 2. The binary order for SQL CHAR columns is different from that of SQL NCHAR columns when the data consists of surrogate pair Unicode characters. As a result, CHAR columns NCHAR columns do not always have the same sort for identical strings.
UTF8	<ol style="list-style-type: none"> 1. You can specify the length of SQL CHAR types as a number of characters. 2. The binary order on the SQL CHAR columns is always the same as that of the SQL NCHAR columns when the data consists of the same surrogate pair Unicode characters. As a result, CHAR columns and NCHAR columns have the same sort for identical strings. 	<ol style="list-style-type: none"> 1. Surrogate pair Unicode characters are stored as 6 bytes instead of the 4 bytes defined by the Unicode standard. As a result, Oracle has to convert data for those surrogate characters.
UTFE	<ol style="list-style-type: none"> 1. Same as UTF8. This is the only Unicode character set for the EBCDIC platform. 	<ol style="list-style-type: none"> 1. Same as UTF8. 2. UTFE is not a standard encoding in the Unicode standard. As a result, clients requiring standard UTF-8 encoding must go through data conversion from UTFE to the standard encoding upon retrieval and insertion.

Table 5–2 Unicode Datatype Solutions

National Character Set	Advantages	Disadvantages
AL16UTF16	<ol style="list-style-type: none"> 1. Asian data in AL16UTF16 is generally more compact than in UTF8. As a result, you will save disk space and have less disk I/O when most of your multilingual data stored in the database is Asian data. 2. Generally speaking, processing strings encoded in the AL16UTF16 character set is faster than those encoded in UTF8 because, in most cases, Oracle9i processes characters in an AL16UTF16 encoded string as fixed-width characters. 3. The maximum length limits for the NCHAR and NVARCHAR2 columns are 1000 and 2000 characters respectively. Because it is fixed-width, the lengths are guaranteed. 	<ol style="list-style-type: none"> 1. European ASCII data requires more disk space to store in AL16UTF16 than in UTF8. If most of your data is European data, the disk space usage is not as efficient as using UTF8. 2. The maximum length limits for NCHAR and NVARCHAR2 are 1000 and 2000 characters, which is less than those for NCHAR (2000) and NVARCHAR2 (4000) in UTF8.
UTF8	<ol style="list-style-type: none"> 1. European data in UTF8 is generally more compact than in AL16UTF16. As a result, you will save disk space and have better response time when most of your multilingual data stored in the database is European data. 2. The maximum length limits for the NCHAR and NVARCHAR2 columns are 2000 and 4000 characters respectively, which is more than those for NCHAR (1000) and NVARCHAR2 (2000) in AL16UTF16. Although the maximum length of the NCHAR and NVARCHAR2 columns are larger in UTF8, the actual storage size is still bound by the byte limits of 2000 and 4000 bytes respectively. For example, you can store 4000 UTF8 characters in an NVARCHAR2 column if all the characters are single byte, but only 4000/3 characters if all the characters are three bytes. 	<ol style="list-style-type: none"> 1. Asian data requires more disk space to store in UTF8 than in AL16UTF16. If most of your data is Asian data, the disk space usage is not as efficient as using AL16UTF16. 2. Although you can specify larger length limits for NCHAR and NVARCHAR, you are not guaranteed to be able to insert the number of characters specified by these limits. This is because it is variable-width. 3. Processing strings encoded in UTF8 is generally slower than those encoded in AL16UTF16 because UTF8 encoded strings consist of variable-width characters.

Unicode Case Studies

This section describes a few typical scenarios for storing Unicode characters in an Oracle9i database.

Example 5–3 Unicode Solution with a Unicode Database

An American company running a Java application would like to add German and French support in the next release of their application, and add Japanese support at a later time. The company currently has the following system configuration:

- The existing database has a database character set of US7ASCII.
- All character data in the existing database is composed of ASCII characters.
- PL/SQL stored procedures are used in the database.
- The database is around 300 GB.
- There is a nightly downtime of 4 hours.

In this case, a typical solution is to choose UTF8 for the database character set because of the following reasons:

- The database is very large and the scheduled downtime is short. How fast the database can be migrated to support Unicode is vital. Because the database is in US7ASCII, the easiest and fastest way of enabling the database to support Unicode is to switch the database character set to UTF8 by issuing the `ALTER DATABASE` statement. No data conversion is required because US7ASCII is a subset of UTF8.
- Because most of the code is written in Java and PL/SQL, changing the database character set to UTF8 is unlikely to break existing code. Unicode support will be automatically enabled in the application.
- Because the application will support French, German, and Japanese, there will be few surrogate characters. Both AL32UTF8 and UTF8 are suitable.

Example 5–4 Unicode Solution with Unicode Datatypes

A European company running its applications mainly on Windows platforms wants to add new Windows applications written in Visual C/C++, which will use the existing database to support Japanese and Chinese customer names. The company currently has the following system configuration:

- The existing database has a database character set of WE8ISO8859P1.

- All character data in the existing database is composed of Western European characters.
- The database is around 50 GB.

In this case, a typical solution is to use `NCHAR` and `NVARCHAR2` datatypes to store Unicode characters, and keep `WE8ISO8859P1` as the database character set and use `AL16UTF16` as the national character set. The reasons for this are:

- Migrating the existing database to a Unicode database required data conversion because the database character set is `WE8ISO8859P1` (a Latin 1 character set), which is not a subset of UTF8. As a result, there will be some overhead in converting the data to UTF8.
- The additional languages are supported in new applications only, so there is no dependency on the existing applications or schemas. It is simpler to use the Unicode datatype in the new schema and keep the existing schemas unchanged.
- Only customer name columns require Unicode support. Using a single `NCHAR` column meets the customer's requirements without migrating the entire database.
- Because the languages to be supported are mostly Asian languages, `AL16UTF16` should be used as the national character set so that disk space is used more efficiently.
- Lengths are treated in terms of characters in the SQL `NCHAR` datatypes. This is the same as the way they are treated when using `wchar_t` strings in Windows C/C++ programs. This reduces programming complexity.
- Existing applications using the existing schemas are unaffected.

Example 5-5 Unicode Solution with Both a Unicode Database and Unicode Datatypes

A Japanese company wants to develop a new Java application on Oracle9i. The company projects that the application will support as many languages as possible in the long run.

- In order to store documents as is, the company decided to use the `BLOB` datatype to store documents of multiple languages.
- The company may also want to generate UTF-8 XML documents from the relational data for business-to-business data exchange.
- The back-end has Windows applications written in C/C++ using ODBC to access the Oracle database.

In this case, the typical solution is to create a Unicode database using AL32UTF8 as the database character set and use the SQL `NCHAR` datatypes to store multilingual data. The national character set should be set to AL16UTF16. The reasons for this solution are:

- When documents of different languages are stored as `BLOBS`, Oracle Text requires the database character set to be one of the UTF-8 character sets. Because the applications may retrieve relational data as UTF-8 XML format (where surrogate characters are stored as four bytes), AL32UTF8 should be used as the database character set to avoid redundant data conversion when UTF-8 data is retrieved or inserted.
- Because applications are new and written in both Java and Windows C/C++, the company should use the SQL `NCHAR` datatype for its relational data as both Java and Windows support the UTF-16 character datatype and the length of a character string is always measured in the number of characters.
- If most of the data is for Asian languages, AL16UTF16 should be used in conjunction with the SQL `NCHAR` datatypes because AL16UTF16 offers better performance and storage efficiency.

Migrating Data to Unicode

It is important to separate the task of character set migration from the task of database version migration. For example, if you have an Oracle8i non-Unicode database and you want to migrate it to an Oracle9i Unicode database, you must first migrate it to Oracle9i, then migrate the data to Unicode.

This section describes how to migrate your data to Unicode in Oracle9i. ["Migrating to Use the NCHAR Datatypes"](#) on page 5-16 describes how to migrate non-Unicode SQL `CHAR` datatypes to SQL `NCHAR` datatypes. It also describes how to migrate pre-Oracle9i SQL `NCHAR` datatypes to Oracle9i SQL `NCHAR` datatypes.

Before you actually migrate your data to Unicode, you need to identify areas of possible data character set conversions and truncation of data. Oracle strongly recommends that you analyze your database using the Character Set Scanner Utility for possible problems before actually migrating the database.

See Also: [Chapter 10, "Character Set Scanner Utility"](#)

Migrating to a Unicode Database

There are three general approaches when migrating data from non-Unicode character set to Unicode:

- [Full Export and Import](#)
- [The ALTER DATABASE CHARACTER SET Statement](#)
- [The ALTER DATABASE CHARACTER SET Statement and Selective Imports and Exports](#)

Full Export and Import

In most cases, a full export and import is recommended to properly convert all data to a Unicode character set. It is important to be aware of data truncation issues because character datatype columns might need to be extended before importing to handle the increase of data byte size.

The steps to migrate to Unicode using a full export and import are:

1. Scan the database to identify columns that need to be extended. Use the Character Set Scanner Utility.

See Also: [Chapter 10, "Character Set Scanner Utility"](#)

2. Export the entire database.
3. Create a new database using either UTF8 or AL32UTF8 on ASCII-based platforms, using UTFE on EBCDIC platforms.
4. Create the tables identified in step 1 with extended columns size.
5. Import the.DMP file exported in step 2 to the new database.

The ALTER DATABASE CHARACTER SET Statement

If, and only if, the current database character set is US7ASCII and all the data is in the 7-bit range, you can use the ALTER DATABASE CHARACTER SET statement to expedite migration to a Unicode database character set. Note that this approach cannot be taken on EBCDIC platforms because UTFE is not a strict superset of any EBCDIC character set.

Use the following steps to migrate to Unicode using the ALTER DATABASE CHARACTER SET statement:

1. Scan the database to make sure all data is in the 7-bit range. Use the Character Set Scanner Utility.

See Also: [Chapter 10, "Character Set Scanner Utility"](#)

2. Change the database character set to UTF8 or AL32UTF8 using the ALTER DATABASE CHARACTER SET statement.

See Also: ["Changing the Character Set After Database Creation"](#) on page 2-15 for the steps to change the database character set

The ALTER DATABASE CHARACTER SET Statement and Selective Imports and Exports

Another approach is to issue an ALTER DATABASE CHARACTER SET statement followed by selective imports. This methods can be used when the distributions of convertible data are known and they are stored within a small number of tables.

The steps to migrate to Unicode using selective imports are:

1. Scan the database to identify tables that contain convertible data.
2. Export those tables identified in step 1.
3. Delete all rows from those table identified in step 1.
4. Change the database character set to UTF8 or AL32UTF8 with the ALTER DATABASE CHARACTER SET statement.
5. Import the dump files into the database.

Migrating to Use the NCHAR Datatypes

The Oracle Server introduced in release 8.0 a national character (NCHAR) datatype that allows for a second, alternate character set in addition to the original database character set. NCHAR supports a number of special, fixed-width Asian character sets that were introduced to provide for higher performance processing of Asian character data.

In Oracle9i, the SQL NCHAR datatypes are limited to the Unicode character set encoding (UTF8 and AL16UTF16) only. Any other Oracle8 Server character sets that were available under the NCHAR datatype, including Asian character sets (for example, JA16SJISFIXED), will no longer be supported.

The migration steps for existing NCHAR, NVARCHAR, and NCLOB columns through export and import are as follows:

1. Export all SQL NCHAR columns from the Oracle8 or Oracle8i database.
2. Drop the SQL NCHAR columns.

3. Upgrade database to Oracle9i.
4. Import the SQL NCHAR columns into Oracle9i.

The Oracle9i migration utility can also convert your Oracle8 and Oracle8i NCHAR columns to 9i NCHAR columns. A SQL NCHAR upgrade script called `utlchar.sql` is supplied with the migration utility. You should run it at the end of the migration to convert your Oracle8 and Oracle8i NCHAR columns to the new Oracle9i NCHAR columns. Once the script has been executed the data cannot be downgraded, because there is no downgrade SQL NCHAR script. The only way for you move back to Oracle8 or Oracle8i is to drop all NCHAR columns, downgrade the database, and import the old NCHAR data from a previous Oracle8 or Oracle8i export file. Make sure you have a backup (export file) of your Oracle8 or Oracle8i NCHAR data, in case you need to downgrade your database in the future.

To take advantage of the new Unicode NCHAR datatypes, you can also use the Export and Import utilities to migrate SQL CHAR columns to SQL NCHAR columns:

1. Export the SQL CHAR columns that you want to convert to SQL NCHAR.
2. Drop the columns that were just exported.
3. Import the columns as SQL NCHAR columns.

See Also:

- *Oracle9i Database Utilities* for a description of export and import procedures
- *Oracle9i Database Migration* for NCHAR migration information

Designing Database Schemas to Support Multiple Languages

In addition to choosing a Unicode solution, the following should also be taken into consideration when the database schema is designed to support multiple languages:

- [Specifying Column Limits](#)
- [Storing Data of Multiple Languages](#)
- [Storing Documents in LOBs](#)

Specifying Column Limits

When you use NCHAR and NVARCHAR2 datatypes for storing multilingual data, the column limit specified for a column is always in character semantics (which is in

terms of the number of Unicode code units). The following table shows the maximum size of the NCHAR and NVARCHAR2 datatypes for the ALT16UTF16 and UTF8 national character sets.

National Character Set	Maximum Size of NCHAR Datatype	Maximum Size of NVARCHAR2 Datatype
ALT16UTF16	1000 characters	2000 characters
UTF8	2000 bytes	4000 bytes

When you use CHAR and VARCHAR2 datatypes for storing multilingual data, the column limit specified for each column is, by default, in number of bytes. If the database needs to support Thai, Arabic, or multibyte languages such as Chinese and Japanese, the limits for the CHAR, VARCHAR, and VARCHAR2 columns may need to be extended. This is because the number of bytes required to encode these languages in UTF8 or AL32UTF8 may be significantly larger than those for English and Western European languages. For example, one Thai character in the Thai character set requires 3 bytes in UTF8 or AL32UTF8. In addition, the maximum limits for CHAR, VARCHAR, and VARCHAR2 datatypes are 2000 bytes, 4000 bytes, and 4000 bytes respectively. If applications need to store more than 4000 bytes, you should use the CLOB datatype for the data.

Storing Data of Multiple Languages

The Unicode character set includes characters of most written languages around the world, but it does not tell you the language to which a given character belongs. In other words, a character such as ä does not contain information about whether it is a French or German character. In order to provide information in the language a user desires, data stored in a Unicode database should accompany the language information to which the data belongs.

There are many ways for a database schema to relate data to a language. Here is one example.

Store Language Information with the Data

For data such as product descriptions or product names, you can add a language column (language_id) of CHAR or VARCHAR2 datatype to the product table to identify the language of the corresponding product information. This enables accessing applications to retrieve the information in the desired language. The possible values for this language column are the 3-letter abbreviations of the valid NLS_LANGUAGE values of the database.

See Also: [Appendix A, "Locale Data"](#), for a list of NLS_LANGUAGE values and their abbreviations

You can also create a view to select the data of the current language. For example:

```
CREATE OR REPLACE VIEW product AS
  SELECT product_id, product_name
  FROM   products_table
  WHERE  language_id = sys_context('USERENV', 'LANG');
```

Select Translated Data Using Fine-Grained Access Control

Fine-grained access control allows you to limit the degree to which a user can view information in a table or view. Typically, this is done by appending a WHERE clause. Once you add a WHERE clause as a fine-grained access policy to a table or view, Oracle9i automatically appends the WHERE clause to any SQL statements on the table at run time so that only those rows satisfying the WHERE clause can be accessed.

You can use this feature to avoid specifying the desired language of an user in the WHERE clause in each and every SELECT statement in your applications. The following WHERE clause limits the view of a table to the rows corresponding to the desired language of a user:

```
WHERE language_id = sys_context('userenv', 'LANG')
```

When you specify this WHERE clause as a fine-grained access policy for your product_table as follows:

```
DBMS_RLS.ADD_POLICY ('scott', 'product_table', 'lang_policy', 'scott',
  'language_id = sys_context('userenv', 'LANG')', 'select');
```

Then any SELECT statement on the table product_table will automatically append the WHERE clause.

Storing Documents in LOBs

You can store documents in multiple languages in CLOB, NCLOB or BLOB and set up Oracle Text to enable content search for the documents.

1. Data in CLOB columns is always stored as UTF-16 internally when the database character set is of varying width, such as UTF8 or AL32UTF8. Document contents are converted to UTF-16 when they are inserted into a CLOB column. This means that the storage space required for an English document doubles

when the data is converted. Storage for an Asian language document, such as Japanese, in a CLOB column requires less storage space than the same document in a LONG column using UTF8 (typically around 30% less, depending on the contents of the document).

- 2. Documents in NCLOB are also stored as UTF-16 regardless of the database character set or national character set. The storage space requirement is the same as in CLOB. Document contents are converted to UTF-16 when they are inserted into a NCLOB column. If you want to store multilingual documents in a non-Unicode database, you should choose NCLOB. However, content search on NCLOB is not yet supported.
- 3. Documents in BLOB format are stored as they are. No data conversion occurs during insert and retrieval. However, SQL string manipulation functions (such as LENGTH or SUBSTR) and collation functions (such as NLS_SORT and ORDER BY) are not applicable to the BLOB datatype.

The following table lists the advantages and disadvantages for datatypes when storing documents:

Table 5–3 Comparison of Datatypes for Document Storage

Datatypes	Advantages	Disadvantages
CLOB	<ul style="list-style-type: none">1. Content search support2. String manipulation support	<ul style="list-style-type: none">1. Dependent on database character set2. Data conversion3. Cannot store binary documents
NCLOB	<ul style="list-style-type: none">1. Independent on database character set2. String manipulation support	<ul style="list-style-type: none">1. No content search support2. Data conversion3. Cannot store binary documents
BLOB	<ul style="list-style-type: none">1. Independent on database character set2. Content search support3. No data conversion, data store as is4. Can store binary documents such as Word or Excel	<ul style="list-style-type: none">1. No string manipulation support

Creating Indexes for Document Content Search

Oracle Text allows you to build indexes for content search on multilingual documents stored as CLOBs and BLOBs. It uses a language-specific lexer to parse the CLOB or BLOB data and produces a list of searchable keywords.

You need to create a multi-lexer for multilingual document searching to work. The multi-lexer chooses a language-specific lexer for each row, based on a language column. This section describes the high level steps to create indexes for documents in multiple languages.

See Also: *Oracle Text Reference*

Creating Multi-Lexers

The first step in creating the multi-lexer is the creation of language-specific lexer preferences for each language supported. The following example creates English, French, and Japanese lexers with PL/SQL procedures:

```
ctx_ddl.create_preference('english_lexer', 'basic_lexer');
ctx_ddl.set_attribute('english_lexer', 'index_themes', 'yes');
ctx_ddl.create_preference('german_lexer', 'basic_lexer');
ctx_ddl.set_attribute('german_lexer', 'composite', 'german');
ctx_ddl.set_attribute('german_lexer', 'alternate_spelling', 'german');
ctx_ddl.set_attribute('german_lexer', 'mixed_case', 'yes');
ctx_ddl.create_preference('japanese_lexer', 'JAPANESE_VGRAM_LEXER');
```

Once the language-specific lexer preferences are created, they need to be gathered together under a single multi-lexer preference. First, create the multi-lexer preference, using the MULTI_LEXER object:

```
ctx_ddl.create_preference('global_lexer', 'multi_lexer');
```

Now we must add the language-specific lexers to the multi-lexer preference using the add_sub_lexer call:

```
ctx_ddl.add_sub_lexer('global_lexer', 'german', 'german_lexer');
ctx_ddl.add_sub_lexer('global_lexer', 'japanese', 'japanese_lexer');
ctx_ddl.add_sub_lexer('global_lexer', 'default', 'english_lexer');
```

This nominates the `german_lexer` preference to handle German documents, the `japanese_lexer` preference to handle French documents, and the `english_lexer` preference to handle everything else, using `DEFAULT` as the language.

Building Indexes for Documents Stored as CLOBs

The multi-lexer decides which lexer to use for each row based on a language column. This is a character column in the table which stores the language of the document in the text column. You should use the Oracle language name to identify the language of a document in this column. For instance, if you use CLOBs to store your documents, then you must add the language column to the table where the documents are stored:

```
CREATE TABLE globaldoc (  
  doc_id      NUMBER          PRIMARY KEY,  
  language    VARCHAR2(30),  
  text        CLOB  
);
```

To create an index for this table, use the multi-lexer preference and specify the name of the language column:

```
CREATE INDEX globalx ON globaldoc(text)  
  indextype IS ctxsys.context  
  parameters ('lexer global_lexer  
              language column language');
```

Creating Indexes for Documents Stored as BLOBs

In addition to the language column, the character set and format columns must be added in the table where your documents are stored. The character set column stores the character set of the documents using the Oracle character set names. The format column specifies whether a document is a text or binary document. For instance, your table would look like:

```
CREATE TABLE globaldoc (  
  doc_id      NUMBER          PRIMARY KEY,  
  language    VARCHAR2(30),  
  characterset VARCHAR2(30),  
  format      VARCHAR2(10),  
  text        BLOB  
);
```

With the format column, you may put word-processing or spreadsheet documents into the table and specify `binary` in the format column. For text documents such as HTML, XML and text, you may put them into the table and specify `text` in the format column. With the character set column, you can store text documents in different character sets.

When you create the index, specify the names of the format and character set columns:

```
CREATE INDEX globalx ON globaldoc(text)
  indextype is ctxsys.context
  parameters ('filter inso_filter
              lexer global_lexer
              language column language
              format column format
              charset column characterset');
```

You may use the `charset_filter` if all documents are in text format.

Unicode Programming

This chapter illustrates programming issues when dealing with Unicode. It contains the following topics:

- [Overview of Unicode Programming](#)
- [SQL and PL/SQL Programming with Unicode](#)
- [OCI Programming with Unicode](#)
- [Pro*C/C++ Programming with Unicode](#)
- [JDBC and SQLJ Programming with Unicode](#)
- [ODBC and OLEDB Programming with Unicode](#)

Overview of Unicode Programming

Oracle9i offers several database access products for inserting and retrieving Unicode data. Oracle offers database access products for most commonly used programming environments such as Java and C/C++. Data is transparently converted between the database and client programs, which ensures that client programs are independent of the database character set and national character set. In addition, client programs are sometimes even independent of the character datatype, such as NCHAR or CHAR, used in the database.

To avoid overloading the database server with data conversion operations, Oracle9i always tries to move them to the client side database access products. In a few cases, data must be converted in the database, and you should be aware of the performance implications. Details of the data conversion paths taken are discussed in this chapter.

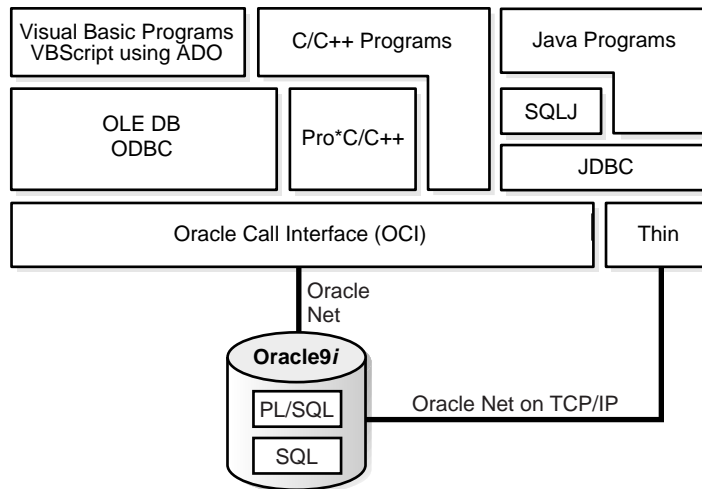
Database Access Product Stack and Unicode

Oracle Corporation offers a comprehensive set of database access products that allow programs from different development environments to access Unicode data stored in the database. These products are listed in [Table 6–1](#).

Table 6–1 Unicode Programming Environments

Programming Environment	Oracle Offers
C/C++	Oracle Call Interface (OCI) Oracle Pro*C/C++ Oracle ODBC Driver Oracle OLE DB Driver
Visual Basic	Oracle ODBC Driver Oracle OLE DB Driver
Java	Oracle JDBC OCI Driver Oracle SQLJ
PL/SQL	Oracle PL/SQL and SQL

[Figure 6–1](#) shows how the database access products can access the database.

Figure 6–1 Oracle Database Access Products

1. The Oracle Call Interface (OCI) is the lowest level API that the rest of the client-side database access products use. It provides a flexible way for C/C++ programs to access Unicode data stored in SQL CHAR and NCHAR datatypes. Using OCI, you can programmatically specify the character set (UTF-8, UTF-16, and others) for the data to be inserted or retrieved.
2. Oracle Pro*C/C++ allows you to embed SQL and PL/SQL in your programs. It uses OCI's Unicode capabilities to provide UTF-16 and UTF-8 data access for SQL CHAR and NCHAR datatypes.
3. The Oracle ODBC driver enables C/C++, Visual Basic, and VBScript programs running on Windows platforms to access Unicode data stored in SQL CHAR and NCHAR datatypes of the database. It provides UTF-16 data access by implementing the SQLWCHAR interface specified in the ODBC standard specification.
4. The Oracle OLE DB driver enables C/C++, Visual Basic, and VBScript programs running on Windows platforms to access Unicode data stored in SQL CHAR and NCHAR datatypes. It provides UTF-16 data access through wide string OLE DB datatypes.
5. Oracle JDBC drivers are the primary Java programmatic interface for accessing an Oracle9i database. Oracle provides two client-side JDBC drivers. The JDBC OCI driver that is used by Java applications and required Oracle OCI library,

the JDBC Thin driver is a pure Java driver that is primarily used by Java applets and only support the Oracle Net protocol over TCP/IP. Both drivers support Unicode data access to SQL `CHAR` and `NCHAR` datatypes in the database.

6. Oracle SQLJ acts like a preprocessor that translates embedded SQL in a Java program into a Java source file with JDBC calls. It offers you a higher level programmatic interface to access databases. Like JDBC, SQLJ provides Unicode data access to SQL `CHAR` and `NCHAR` datatypes in the database.
7. The PL/SQL and SQL engines process PL/SQL programs and SQL statements on behalf of client-side programs such as OCI and server-side PL/SQL stored procedures. They allow PL/SQL programs to declare `NCHAR` and `NVARCHAR2` variables and access SQL `NCHAR` datatypes in the database.

The following sections describe how each of the above database access products supports Unicode data access to an Oracle9i database and offer examples for using those products.

SQL and PL/SQL Programming with Unicode

SQL is the fundamental language with which all programs and users access data in an Oracle database either directly or indirectly. PL/SQL is a procedural language that combines the data manipulating power of SQL with the data processing power of procedural languages. Both SQL and PL/SQL can be embedded in other programming languages. This section describes Unicode-related features in SQL and PL/SQL that you can deploy for multilingual applications.

SQL `NCHAR` Datatypes

There are three SQL `NCHAR` datatypes:

- `NCHAR`
- `NVARCHAR2`
- `NCLOB`

When you define a table column or PL/SQL variables in `NCHAR`, the length specified is always in the number of characters. For example, the following statement:

```
CREATE TABLE tabl (col1 NCHAR(30));
```

creates a column with a maximum character length of 30. The maximum byte length is the multiple of the maximum character length and the maximum number of bytes

per character. For example, if the national character set is UTF8, the above statement defines a maximum byte length of 90 bytes.

The national character set is defined when the database is created. In Oracle9i, the national character set can be either UTF8 or AL16UTF16.

You can define a maximum column size of 2000 characters when the national character set is UTF8 and 1000 when it is AL16UTF16. The actual data is subject to the maximum byte limit of 2000. The two size constraints must be satisfied at the same time. In PL/SQL, the maximum length of NCHAR data is 32767 bytes. You can define an NCHAR variable of up to 32767 characters, but the actual data cannot exceed 32767 bytes. If you insert a value that is shorter than the column length, Oracle blank pads the value to the smaller value between maximum character length and maximum byte length.

The NVARCHAR2 datatype specifies a variable length national character set character string. When you create a table with an NVARCHAR2 column, you supply the maximum number of characters. Oracle subsequently stores each value in the column exactly as you specify it, provided the value does not exceed the column's maximum length. It does not pad the string value to the maximum length. Lengths for NVARCHAR2 are always treated as being in units of characters, just as for NCHAR.

The maximum column size allowed is 4000 characters when the national character set is UTF8 and 2000 when it is AL16UTF16. NVARCHAR2 columns can be defined up to 4000 bytes, the actual maximum length of a column allowed is the number of characters that fit into no more than 4000 bytes. In PL/SQL, the maximum length for NVARCHAR2 is 32767 bytes. You can define NVARCHAR2 variables up to 32767 characters, but the actual data cannot exceed 32767 bytes.

The following statement creates a table with one NVARCHAR2 column of 2000 characters in length. If the national character set is UTF8, the following will create a column with maximum character length of 2000 and maximum byte length of 4000.

```
CREATE TABLE tabl (coll NVARCHAR2(2000));
```

NCLOB is a character large object containing multibyte characters, with a maximum size of 4 gigabytes. Unlike BLOBs, NCLOBs have full transactional support so changes made through SQL, the DBMS_LOB package, or OCI participate fully in transactions. NCLOB value manipulations can be committed and rolled back. Note, however, that you cannot save an NCLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

NCLOB values are stored in the database using the UTF-16 character set, which has a fixed width. Oracle translates the stored Unicode value to the character set requested on the client or on the server, which can be fixed-width or variable-width.

When you insert data into an NCLOB column using a varying-width character set, Oracle converts the data into UTF-16 Unicode before storing it in the database. This happens whether the national character set is UTF8 or AL16UTF16.

Implicit Conversion

Oracle supports implicit conversions between SQL NCHAR datatypes and most Oracle datatypes, such as CHAR, VARCHAR2, NUMBER, DATE, ROWID, and CLOB. Any implicit conversions for CHAR/VARCHAR2 are also supported for SQL NCHAR datatypes. You can use SQL NCHAR datatypes the same way as SQL CHAR datatypes.

There are several points to keep in mind with implicit conversions:

- Type conversions between SQL CHAR datatypes and SQL NCHAR datatypes may involve character set conversion when database and national character sets are different, or blank padding if the target data is either CHAR or NCHAR.
- Implicit conversion between CLOB and NCLOB datatypes is not possible. You can, however, use Oracle's explicit conversion functions for them.

Exception Handling for Data Loss

Data loss can occur during type conversion when character set conversion is necessary. If a character in the first character set is not defined in the target character set, then a replacement character will be used in its place. For example, if you try to insert NCHAR data into a regular CHAR column, if the character data in NCHAR (Unicode) form cannot be converted to the database character set, the character will be replaced by a replacement character question mark. The NLS_NCHAR_CONV_EXCP initialization parameter controls the behavior of data loss during character type conversion. When this parameter is set to TRUE, any SQL statements that result in data loss return an ORA-12713 error and the corresponding operation is aborted. When this parameter is set to FALSE, data loss is not reported and the unconvertible characters are replaced with replacement characters. The default value is TRUE. This parameter works for both implicit and explicit conversion.

In PL/SQL, when data loss occurs during conversion of SQL CHAR and NCHAR datatypes, the exception LOSSY_CHARSET_CONVERSION is raised. It applies for both implicit and explicit conversion.

Direction Rules for Implicit Type Conversion

In some cases, conversion is only possible in one direction. In other cases, both directions are possible. In order to have predictable and unambiguous behavior, Oracle defines a set of specific rules for conversion direction.

- **INSERT/UPDATE statement**

Values are converted to the type of target database column.

- **SELECT INTO statement**

Data from the database is converted to the type of target variable.

- **Assignments**

Values on the right hand side of (=) are converted to the types of the variable on left hand side which is the target of assignment.

- **Parameters to SQL and PL/SQL functions**

CHAR, VARCHAR2, NCHAR, and NVARCHAR2 overload in the same way. An argument of one with a CHAR, VARCHAR2, NCHAR or NVARCHAR2 type will match a formal parameter of any of the types CHAR, VARCHAR2, NCHAR or NVARCHAR2 for overloading. If the argument and formal parameter types do not match exactly, then implicit conversions will be introduced when data is copied into the parameter on function entry and copied out to the argument on function exit.

- **Concatenation || operation or CONCAT function**

If one operand is a SQL CHAR or NCHAR datatype and the other operand is a NUMBER or other non-character datatype, the other datatype is converted to VARCHAR2 or NVARCHAR2. For concatenation between character datatypes, see the discussion below.

- **Arithmetic expressions**

- SQL CHAR or NCHAR datatypes and NUMBER: Character value is converted to NUMBER.
- SQL CHAR or NCHAR datatypes and DATE: Character string value is converted to DATE type.
- SQL CHAR or NCHAR datatypes and ROWID: Character types are converted to rowid type.
- For arithmetic operations between SQL NCHAR datatypes and SQL CHAR datatypes, the character data is converted to NUMBER.

- **Comparisons between different datatypes**

- SQL CHAR or NCHAR datatypes and NUMBER
Character values are converted to NUMBER.

- SQL CHAR or NCHAR datatypes and DATE
Character values are converted to DATE.
- SQL CHAR or NCHAR datatypes and ROWID
Character data is converted to ROWID.
- SQL NCHAR datatypes and SQL CHAR datatypes
Comparisons between SQL NCHAR datatypes and SQL CHAR datatypes are more complex because they can be encoded in different character sets. For comparisons between CHAR and VARCHAR2, or between NCHAR and NVARCHAR2, the direction is CHAR->VARCHAR2 or NCHAR->NVARCHAR2. When there is conversion between SQL NCHAR datatypes and SQL CHAR datatypes, character set conversion occurs if they are encoded in different character sets. The character set for SQL NCHAR datatypes is always Unicode and can be either UTF8 or AL16UTF16 encoding, which have equal character repertoires but are different encodings of the Unicode standard. SQL CHAR datatypes use the database character set, which can be any character set that Oracle supports. Unicode is always a superset of any character set supported by Oracle, so it is always convertible from SQL CHAR datatypes to SQL NCHAR datatypes without data loss.

SQL Functions for Unicode Datatypes

SQL NCHAR datatypes can be converted to and from SQL CHAR datatypes and other datatypes using explicit conversion functions. Following are several examples using this table.

```
CREATE TABLE customers
(id NUMBER, name NVARCHAR2(50), addr NVARCHAR2(200), dob DATE);
```

Example 6–1 Unicode Datatype Example 1

```
INSERT INTO customers VALUES (1000,
TO_NCHAR('John Smith'), N'500 Oracle Parkway');
```

Example 6–2 Unicode Datatype Example 2

```
SELECT name FROM customers WHERE TO_CHAR(name) LIKE 'Sm%';
```

Example 6–3 Unicode Datatype Example 3

```
DECLARE
ndstr NVARCHAR2(20) := N'12-SEP-1975';
BEGIN
```

```
SELECT name FROM customers
WHERE (dob)> TO_DATE(ndstr, 'DD-MON-YYYY', N'NLS_DATE_LANGUAGE = AMERICAN');
END;
```

As demonstrated in [Example 6-3](#), not only can SQL NCHAR data be passed to explicit conversion functions, but also SQL CHAR and NCHAR data can be mixed together when using multiple string parameters.

See Also: *Oracle9i SQL Reference* for more information about explicit conversion functions for SQL NCHAR datatypes

Other SQL String Functions

Most SQL functions can take arguments of SQL NCHAR datatypes as well as mixed character datatypes. The return datatype is based on the type of the first argument. If a non-string datatype like NUMBER or DATE is passed to these functions, it will be converted to VARCHAR2. Several examples using the `customers` table from above follow:

Example 6-4 String Function Example 1

```
SELECT INSTR(name, N'Sm', 1, 2) FROM customers;
```

Example 6-5 String Function Example 2

```
SELECT CONCAT(name || id) FROM customer;
```

`id` will be converted to NVARCHAR2 and then concatenated with `name`.

Example 6-6 String Function Example 3

```
SELECT RPAD (name, 100, ' ') FROM customer;
```

Space character ' ' is converted to the corresponding character in the NCHAR character set and then padded to the right of `name` until the total display length reaches 100.

See Also: *Oracle9i SQL Reference* for a list of all SQL functions that can accept SQL NCHAR datatypes

Unicode String Literals

You can input Unicode string literals in SQL and PL/SQL as follows:

- Put a prefix `N` in front of a single quote marked string literal. This explicitly indicates that the following string literals is an `NCHAR` string literal.
- Mark a string literal with single quotations. Because Oracle supports implicit conversions to SQL `NCHAR` datatypes, a string literal is converted to a SQL `NCHAR` datatype wherever necessary.

When a string literal is included in a query and the query is submitted through a client-side tool such as SQL*Plus, all the queries are encoded in the client's character set and then converted to the server's database character set before processing. Therefore, data loss can occur if the string literal cannot be converted to the server database character set.

- Use the `NCHR(n)` SQL function, which returns the character having the binary equivalent to `n` in the national character set, which is UTF8 or AL16UTF16. The result of concatenations of several `NCHR(n)` is `NVARCHAR2`. In this way, you can bypass the client and server character set conversions and create an `NVARCHAR2` string directly. For example, `NCHR(32)` represents a blank character.
- Use the `UNISTR(string)` SQL function. `UNISTR(string)` takes a string and converts it to Unicode. The result is in database national character set (UTF8 or AL16UTF16). You can embed escape `\bbbb` inside the string. The escape represents the value of a UTF-16 code point with hex number `0xbbbb`. For example, `UNISTR('G\0061ry')` represents 'Gary'.

The last two methods can be used to encode any Unicode string literals.

Using the UTL_FILE Package with NCHAR

The `UTL_FILE` package has been enhanced in Oracle9i to handle Unicode national character set data. The following functions and procedures have been added:

- `FOPEN_NCHAR`

This function opens a file in Unicode for input or output, with the maximum line size specified. With this function, you can read or write a text file in Unicode instead of in the database character set.

- `GET_LINE_NCHAR`

This procedure reads text from the open file identified by the file handle and places the text in the output buffer parameter. With this function, you can read a text file in Unicode instead of in the database character set.

- `PUT_NCHAR`

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle. With this function, you can write a text file in Unicode instead of in the database character set.

- `PUT_LINE_NCHAR`

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle. With this function, you can write a text file in Unicode instead of in the database character set.

- `PUTF_NCHAR`

This procedure is a formatted `PUT_NCHAR` procedure. With this function, you can write a text file in Unicode instead of in the database character set.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference* for more information about the `UTL_FILE` package

OCI Programming with Unicode

OCI is the lowest-level API for accessing a database, so it offers the best possible performance. When using Unicode with OCI, you should consider these topics:

- [OCI Unicode Code Conversion](#)
- [When NLS_LANG is Set to UTF8 or AL32UTF8 in OCI](#)
- [Binding and Defining SQL CHAR Datatypes in OCI](#)
- [Binding and Defining SQL NCHAR Datatypes in OCI](#)
- [Binding and Defining CLOB and NCLOB Unicode Data in OCI](#)
- [Unicode Mode in OCI](#)

OCI Unicode Code Conversion

Unicode character set conversions take place between an OCI client and the database server if the client and server character sets are different. The conversion occurs on either the client or the server depending on the circumstances, but usually on the client side.

Data Integrity

You can lose data during conversion if you call an OCI API inappropriately. If the server and client character sets are different, you can lose data when the destination character set is a smaller set than the source character set. You can avoid this

potential problem if both character sets are Unicode character sets (for example, UTF8 and AL16UTF16).

When you bind or define SQL NCHAR datatypes, you should set OCI_ATTR_CHARSET_FORM to SQLCS_NCHAR. Otherwise, you can lose data because the data is converted to the database character set prior to converting to or from the national character set, but only if the database character set is not Unicode.

OCI Performance Implications When Using Unicode

Redundant data conversions can cause performance degradation in your OCI applications. These conversions occur in two cases:

- When you bind or define SQL CHAR datatypes and set the OCI_ATTR_CHARSET_FORM attribute to SQLCS_NCHAR, data conversions take place from client character set to the national database character set, and from the national character set to the database character set. No data loss is expected, but two conversions happen, even though it requires only one.
- When you bind or define SQL NCHAR datatypes and do not set OCI_ATTR_CHARSET_FORM, data conversions take place from client character set to the database character set, and from the database character set to the national database character set. In the worst case, data loss can occur if the database character set is smaller than the client's.

To avoid performance problems, you should always specify the correct form of use based upon the datatype of the target columns. If you do not know the target datatype, you should set the OCI_ATTR_CHARSET_FORM attribute to SQLCS_NCHAR when binding and defining.

Table 6–2 OCI Code Conversions in Unicode

Datatypes for OCI Client Buffer	OCI_ATTR_CHARSET_FORM	Datatypes of the Target Column in the Database	Conversions	Comments
utext	SQLCS_IMPLICIT	CHAR, VARCHAR2, CLOB	UTF-16 to/from database character set in OCI	No unexpected data loss
utext	SQLCS_NCHAR	NCHAR, NVARCHAR2, NCLOB	UTF-16 to/from national character set in OCI	No unexpected data loss
utext	SQLCS_NCHAR	CHAR, VARCHAR2, CLOB	UTF-16 to/from national character set in OCI National character set to/from database character set in database server	No unexpected data loss, but may have performance degradation because the conversion goes through the national character set
utext	SQLCS_IMPLICIT	NCHAR, NVARCHAR2, NCLOB	UTF-16 to/from database character set in OCI Database character set to/from national character set in database server	Data loss may happen because the conversion goes through the database character set
text	SQLCS_IMPLICIT	CHAR, VARCHAR2, CLOB	NLS_LANG character set to/from database character set in OCI	No unexpected data loss
text	SQLCS_NCHAR	NCHAR, NVARCHAR2, NCLOB	NLS_LANG character set to/from national character set in OCI	No unexpected data loss
text	SQLCS_NCHAR	CHAR, VARCHAR2, CLOB	NLS_LANG character set to/from national character set in OCI National character set to/from national character set in database server	No unexpected data loss, but performance degradation may occur because the conversion goes through the national character set
text	SQLCS_IMPLICIT	NCHAR, NVARCHAR2, NCLOB	NLS_LANG character set to/from database character set in OCI Database character set to/from national character set in database server	Data loss may occur because the conversion goes through the database character set

OCI Unicode Data Expansion

Data conversion can result in data expansion, which can cause a buffer to overflow. For binding operations, you need to set the `OCI_ATTR_MAXDATA_SIZE` attribute to a large enough size to hold the expanded data on the server. If this is difficult to do, you need to consider changing the table schema. For defining operations, client applications need to allocate enough buffer for the expanded data. The size of buffer should be the maximum expanded size of data length. You can estimate the maximum buffer length with the following calculation:

1. Get the column data byte size.
2. Multiply it by the maximum number of bytes per character in the client character set.

This method is the simplest and quickest way, but may not be accurate and can waste memory. It is applicable to any character set combination. For example, for UTF-16 data binding and defining, the following example calculates the client buffer:

```
ub2 csid = OCI_UTF16ID;
oratext *selstmt = "SELECT ename FROM emp";
counter = 1;
...
OCISTmtPrepare(stmthp, errhp, selstmt, (ub4)strlen((char*)selstmt),
               OCI_NIV_SYNTAX, OCI_DEFAULT);
OCISTmtExecute ( svchp, stmthp, errhp, (ub4)0, (ub4)0,
                 (CONST OCISnapshot*)0, (OCISnapshot*)0,
                 OCI_DESCRIBE_ONLY);
OCIParamGet(stmthp, OCI_HTYPE_STMT, errhp, &myparam, (ub4)counter);
OCIAttrGet((void*)myparam, (ub4)OCI_DTYPE_PARAM, (void*)&col_width,
            (ub4*)0, (ub4)OCI_ATTR_DATA_SIZE, errhp);
...
maxenamelen = (col_width + 1) * sizeof(utext);
cbuf = (utext*)malloc(maxenamelen);
...
OCIDefineByPos(stmthp, &dfnp, errhp, (ub4)1, (void *)cbuf,
               (sb4)maxenamelen, SQLT_STR, (void *)0, (ub2 *)0,
               (ub2*)0, (ub4)OCI_DEFAULT);
OCIAttrSet((void *) dfnp, (ub4) OCI_HTYPE_DEFINE, (void *) &csid,
            (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCISTmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT);
...
```

When NLS_LANG is Set to UTF8 or AL32UTF8 in OCI

You can use UTF8 and AL32UTF8 by setting NLS_LANG for OCI client applications. If you do not need surrogate characters, it does not matter whether you choose UTF8 or AL32UTF8. However, if your OCI applications might handle surrogate characters, you need to make a decision. Because UTF8 can require up to three bytes per character, one surrogate character is represented in two codepoints, totalling six bytes. With AL32UTF8, one surrogate character is represented in one codepoint, totalling four bytes.

Do not set NLS_LANG to AL16UTF16, because AL16UTF16 is the national character set for the server. If you need to use UTF-16, you should specify the client character set to OCI_UTF16ID using OCIAttrSet when binding or defining data.

Binding and Defining SQL CHAR Datatypes in OCI

To specify a Unicode character set for binding and defining data with SQL CHAR datatypes, you may need to call the OCIAttrSet function to set the appropriate character set ID after OCIBind or OCIDefine APIs. There are two typical cases:

1. Call OCIBind / OCIDefine followed by OCIAttrSet to specify UTF-16 Unicode character set encoding.

```
...
ub2 csid = OCI_UTF16ID;
utext ename[100]; /* enough buffer for ENAME */
...
/* Inserting Unicode data */
OCIBindByName(stmthp1, &bndlp, errhp, (oratext*)" :ENAME",
              (sb4)strlen((char*)" :ENAME"), (void *) ename, sizeof(ename),
              SQLT_STR, (void *)&insname_ind, (ub2 *) 0, (ub2 *) 0, (ub4) 0,
              (ub4 *)0, OCI_DEFAULT);
OCIAttrSet((void *) bndlp, (ub4) OCI_HTYPE_BIND, (void *) &csid,
           (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAttrSet((void *) bndlp, (ub4) OCI_HTYPE_BIND, (void *) &ename_col_len,
           (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving Unicode data */
OCIDefineByPos (stmthp2, &dfnlp, errhp, (ub4)1, (void *)ename,
               (sb4)sizeof(ename), SQLT_STR, (void *)0, (ub2 *)0,
               (ub2*)0, (ub4)OCI_DEFAULT);
OCIAttrSet((void *) dfnlp, (ub4) OCI_HTYPE_DEFINE, (void *) &csid,
           (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
...
```

If bound buffers are of the `utext` datatype, you should add a cast (`text*`) when `OCIBind` or `OCIDefine` is called. The value of the `OCI_ATTR_MAXDATA_SIZE` attribute is usually determined by the size of column on the server character set because this size is only used to allocate temporary buffer for conversion on the server when you perform binding operations.

2. Call `OCIBind` or `OCIDefine` with `NLS_LANG` set to `UTF8` or `AL32UTF8`.

`UTF8` or `AL32UTF8` can be set in `NLS_LANG`. You call `OCIBind` and `OCIDefine` in exactly the same manner as when you are not using Unicode. Set the environment variable `NLS_LANG` to `UTF8` or `AL32UTF8` and run the following OCI program:

```
...
oratext ename[100]; /* enough buffer size for ENAME */
...
/* Inserting Unicode data */
OCIBindByName(stmthp1, &bndlp, errhp, (oratext*)" :ENAME",
              (sb4)strlen((char *)":ENAME"), (void *) ename, sizeof(ename),
              SQLT_STR, (void *)&insname_ind, (ub2 *) 0, (ub2 *) 0,
              (ub4) 0, (ub4 *)0, OCI_DEFAULT);
OCIAttrSet((void *) bndlp, (ub4) OCI_HTYPE_BIND, (void *) &ename_col_len,
           (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving Unicode data */
OCIDefineByPos (stmthp2, &dfnlp, errhp, (ub4)1, (void *)ename,
                (sb4)sizeof(ename), SQLT_STR, (void *)0, (ub2 *)0, (ub2*)0,
                (ub4)OCI_DEFAULT);
...
```

Binding and Defining SQL NCHAR Datatypes in OCI

Oracle recommends you access SQL NCHAR datatypes using UTF-16 binding or defining when using OCI. Starting from Oracle9i, SQL NCHAR datatypes have become pure-Unicode datatypes with an encoding of either UTF8 or AL16UTF16. To access data in SQL NCHAR datatypes, you need to set the OCI attribute `OCI_ATTR_CHARSET_FORM` to `SQLCS_NCHAR` after binding and defining until execution so that it performs an appropriate data conversion without data loss. The length of data in SQL NCHAR datatypes is always in the number of Unicode codepoints.

The following program is a typical example of inserting and fetching data against an NCHAR data column:

```
...
ub2 csid = OCI_UTF16ID;
```

```

ub2 cform = SQLCS_NCHAR;
utext ename[100]; /* enough buffer for ENAME */
...
/* Inserting Unicode data */
OCIBindByName(stmthp1, &bndlp, errhp, (oratext*)" :ENAME",
              (sb4)strlen((char *)":ENAME"), (void *) ename,
              sizeof(ename), SQLT_STR, (void *)&insname_ind, (ub2 *) 0,
              (ub2 *) 0, (ub4) 0, (ub4 *)0, OCI_DEFAULT);
OCIAttrSet((void *) bndlp, (ub4) OCI_HTYPE_BIND, (void *) &csid, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAttrSet((void *) bndlp, (ub4) OCI_HTYPE_BIND, (void *) &cform, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_FORM, errhp);
OCIAttrSet((void *) bndlp, (ub4) OCI_HTYPE_BIND, (void *) &ename_col_len,
           (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving Unicode data */
OCIDefineByPos (stmthp2, &dfnlp, errhp, (ub4)1, (void *)ename,
               (sb4)sizeof(ename), SQLT_STR, (void *)0, (ub2 *)0, (ub2*)0,
               (ub4)OCI_DEFAULT);
OCIAttrSet((void *) dfnlp, (ub4) OCI_HTYPE_DEFINE, (void *) &csid, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAttrSet((void *) dfnlp, (ub4) OCI_HTYPE_DEFINE, (void *) &cform, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_FORM, errhp);
...

```

Binding and Defining CLOB and NCLOB Unicode Data in OCI

In order to write (bind) and read (define) UTF-16 data for CLOB or NCLOB columns, the UTF-16 character set ID must be specified as `OCILOBWrite` and `OCILOBRead`. When you write UTF-16 data into a CLOB column, you should call `OCILOBWrite` as follows:

```

...
ub2 csid = OCI_UTF16ID;
err = OCILOBWrite (ctx->svchp, ctx->errhp, lobp, &amtp, offset, (void *) buf,
                  (ub4) BUFSIZE, OCI_ONE_PIECE, (void *)0,
                  (sb4 (*)()) 0, (ub2) csid, (ub1) SQLCS_IMPLICIT);

```

Where the parameter `amtp` is the data length in the number of Unicode codepoints. The parameter `offset` indicates the offset of data from the beginning of data column. The parameter `csid` must be set for UTF-16 data.

To read UTF-16 data from CLOB columns, call `OCILOBRead` as follows:

```

...

```

```
ub2 csid = OCI_UTF16ID;
err = OCILobRead(ctx->svchp, ctx->errhp, lobb, &amtp, offset, (void *) buf,
                (ub4)BUFSIZE, (void *) 0, (sb4 (*)()) 0, (ub2)csid,
                (ub1) SQLCS_IMPLICIT);
```

The data length is always represented in the number of Unicode codepoints. Note one Unicode surrogate character is counted as two codepoints, because the encoding is UTF-16. After binding or defining LOB column, you can measure the data length stored in the LOB column using `OCILobGetLength`. The returning value is the data length in the number of codepoints if you bind or define as UTF-16.

```
err = OCILobGetLength(ctx->svchp, ctx->errhp, lobb, &lenp);
```

If you are using an NCLOB, you must set `OCI_ATTR_CHARSET_FORM` to `SQLCS_NCHAR`.

Unicode Mode in OCI

OCI supports UTF-16 metadata as well as UTF-16 data for binding and defining. SQL statements, usernames, error messages, and column names can be in UTF-16, and are thus independent of the `NLS_LANG` setting. Oracle provides a Unicode mode so you can use UTF-16 metadata. At the beginning of your OCI program, all you have to do is create OCI environment handle (`OCIEnv`) with the `OCI_UTF16` flag. Any inherited handle is automatically set the mode to Unicode where OCI treats all string parameters as UTF-16 data.

To enable Unicode in OCI applications, Oracle offers an alternative approach using a Unicode API called `OCI_UTF16ID`. Before Oracle9i, OCI could only manipulate UTF-16 character set encoding for binding / defining to insert and fetch data against database columns, while metadata like SQL statement, username, and column name were restricted to the character set specified by `NLS_LANG`. For Oracle9i, the Unicode API is intended to be independent of `NLS_LANG`. In addition, all data manipulation by OCI is in the UTF-16 character set encoding. This is especially important for multilingual applications.

You activate the Unicode API by setting a Unicode mode when creating an OCI environment handle (`OCIEnv`). Any inherited handle from the OCI environment handle will be set to Unicode mode automatically. By changing to Unicode mode, all metatext data parameters (`text*`) are assumed to be Unicode text datatypes (`utext*`) in UTF-16 encoding. For binding and defining, the data is also assumed to be Unicode in UTF-16 encoding. For example, the following program shows how

you can create an OCI environment handle as a Unicode mode. `OCI_UTF16` indicates the default character set is UTF-16:

```
OCIEnv *envhp;
status = OCIEnvCreate((OCIEnv **) &envhp,
OCI_UTF16,
(void *)0,
(void (*)(*)()) 0,
(void (*)(*)()) 0,
(void (*)(*)()) 0,
(size_t) 0,
(void **)0);
```

To prepare the SQL statement, call `OCIStmtPrepare` with `(utext*)` string. The following example runs on Windows platforms only. You may need to change `wchar_t` datatypes for other platforms.

```
const wchar_t sqlstr[] = L"SELECT * FROM ENAME=:ename";
...
OCIStmt* stmthp;
sts = OCIHandleAlloc(envh, (void **) &stmthp, OCI_HTYPE_STMT, 0, NULL);
status = OCIStmtPrepare(stmthp, errhp, (const text*)sqlstr, wcslen(sqlstr),
OCI_NIV_SYNTAX, OCI_DEFAULT);
```

You call `OCIStmtPrepare` exactly like other character sets, but the parameter is UTF-16 string data. For binding and defining data, you do not have to set the `OCI_ATTR_CHARSET_ID` attribute because you are already in the Unicode mode. Bind variable names must be UTF-16 strings. You must cast `(text*)` or `(const(text*))` for metadata parameters.

```
/* Inserting Unicode data */
OCIBindByName(stmthp1, &bndlp, errhp, (const text*)L":ename",
(sb4)wcslen(L":ename"),
(void *) ename, sizeof(ename), SQLT_STR, (void *)&insname_ind,
(ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *)0, OCI_DEFAULT);
OCIAttrSet((void *) bndlp, (ub4) OCI_HTYPE_BIND, (void *) &ename_col_len,
(ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving Unicode data */
OCIDefineByPos (stmthp2, &dfnlp, errhp, (ub4)1, (void *)ename,
(sb4)sizeof(ename), SQLT_STR, (void *)0, (ub2 *)0, (ub2*)0,
(ub4)OCI_DEFAULT);
...
```

Then `OCIExecute` will perform the operation.

Pro*C/C++ Programming with Unicode

Pro*C/C++ provides three ways for you to insert or retrieve Unicode data into or from the database:

1. Using the `VARCHAR` Pro*C/C++ datatype or the native C/C++ `text` datatype, a program can access Unicode data stored in SQL `CHAR` datatypes of a UTF8 or AL32UTF8 database. Alternatively, a program could use the C/C++ native `text` type.
2. Using the `UVARCHAR` Pro*C/C++ datatype or the native C/C++ `utext` datatype, a program can access Unicode data stored in `NCHAR` datatypes of a database.
3. Using the `NVARCHAR` Pro*C/C++ datatype, a program can access Unicode data stored in `NCHAR` datatypes. The difference between `UVARCHAR` and `NVARCHAR` in a Pro*C/C++ program is that the data for the `UVARCHAR` datatype is stored in a `utext` buffer while the data for the `NVARCHAR` datatype is stored in a `text` datatype.

Pro*C/C++ does not use the Unicode OCI API for SQL text. As a result, embedded SQL text must be encoded in the character set specified in the `NLS_LANG` environment variable.

Pro*C/C++ Data Conversion in Unicode

Data conversion occurs in the OCI layer, but it is the Pro*C/C++ preprocessor that instructs OCI which conversion path should be taken based on the datatypes used in a Pro*C/C++ program. [Table 6-3](#) illustrates the conversion paths:

Table 6-3 *Pro*C/C++ Bind and Define Data Conversion*

Pro*C/C++ Datatype	SQL Datatype	Conversion Path
<code>VARCHAR</code> or <code>text</code>	<code>CHAR</code>	<code>NLS_LANG</code> character set to/from the database character set happens in OCI
<code>VARCHAR</code> or <code>text</code>	<code>NCHAR</code>	<code>NLS_LANG</code> character set to/from database character set happens in OCI Database character set to/from national character set happens in database server
<code>NVARCHAR</code>	<code>NCHAR</code>	<code>NLS_LANG</code> character set to/from national character set happens in OCI

Table 6–3 Pro*C/C++ Bind and Define Data Conversion (Cont.)

Pro*C/C++ Datatype	SQL Datatype	Conversion Path
NVARCHAR	CHAR	NLS_LANG character set to/from national character set happens in OCI National character set to/from database character set in database server
UVARCHAR or utext	NCHAR	UTF-16 to/from the national character set happens in OCI
UVARCHAR or utext	CHAR	UTF-16 to/from national character set happens in OCI National character set to database character set happens in database server

Using the VARCHAR Datatype

The Pro*C/C++ VARCHAR datatype is preprocessed to a struct with a length field and text buffer field. An example is shown below using the C/C++ text native datatype and the VARCHAR Pro*C/C++ datatypes to bind and define table columns.

```
#include <sqlca.h>
main()
{
    ...
    /* Change to STRING datatype: */
    EXEC ORACLE OPTION (CHAR_MAP=STRING) ;
    text ename[20] ;                /* unsigned short type */
    varchar address[50] ;           /* Pro*C/C++ uvarchar type */

    EXEC SQL SELECT ename, address INTO :ename, :address FROM emp;
    /* ename is NULL-terminated */
    printf("ENAME = %s, ADDRESS = %.*s\n", ename, address.len, address.arr);
    ...
}
```

When you use the VARCHAR datatype or native text datatype in a Pro*C/C++ program, the preprocessor assumes that the program intends to access columns of SQL CHAR datatypes instead of SQL NCHAR datatypes in the database. The preprocessor generates C/C++ code to reflect this fact by doing a bind or define using the SQLCS_IMPLICIT value for the OCI_ATTR_CHARSET_FORM attribute. As a result, if a bind or define variable is bound to a column of SQL NCHAR datatypes in the database, implicit conversion happens in the database server to convert the data from the database character set to the national database character set and vice

versa. During the conversion, data loss occurs when the database character set is a smaller set than the national character set.

Using the NVARCHAR Datatype

The Pro*C/C++ NVARCHAR datatype is similar to the Pro*C/C++ VARCHAR datatype. It should be used to access SQL NCHAR datatypes in the database. It tells Pro*C/C++ preprocessor to bind or define a text buffer to the column of SQL NCHAR datatypes. The preprocessor will specify the SQLCS_NCHAR value for the OCI_ATTR_CHARSET_FORM attribute of the bind or define variable. As a result, no implicit conversion occurs in the database.

If the NVARCHAR buffer is bound against columns of SQL CHAR datatypes, the data in the NVARCHAR buffer (encoded in the NLS_LANG character set) is converted to or from the national character set in OCI, and the data is then converted to the database character set in the database server. Data can be lost when the NLS_LANG character set is a larger set than the database character set.

Using the UVARCHAR Datatype

The UVARCHAR datatype is preprocessed to a struct with a length field and utext buffer field. The following example code contains two host variables, `ename` and `address`. The `ename` host variable is declared as a `utext` buffer containing 20 Unicode characters. The `address` host variable is declared as a `uvarchar` buffer containing 50 Unicode characters, the `len` and `arr` fields are accessible as fields of a struct.

```
#include <sqlca.h>
#include <sqlucs2.h>

main()
{
    ...
    /* Change to STRING datatype:      */
    EXEC ORACLE OPTION (CHAR_MAP=STRING) ;
    utext ename[20] ;                      /* unsigned short type */
    uvarchar address[50] ;                 /* Pro*C/C++ uvarchar type */

    EXEC SQL SELECT ename, address INTO :ename, :address FROM emp;
    /* ename is NULL-terminated */
    wprintf(L"ENAME = %s, ADDRESS = %.*s\n", ename, address.len,
address.arr);
    ...
}
```

When you use the `UVARCHAR` datatype or native `utext` datatype in Pro*C/C++ programs, the preprocessor assumes that the program intends to access SQL `NCHAR` datatypes. The preprocessor generates C/C++ code by binding or defining using the `SQLCS_NCHAR` value for `OCI_ATTR_CHARSET_FORM` attribute. As a result, if a bind or define variable is bound to a column of a SQL `NCHAR` datatype, an implicit conversion of the data from the national character set occurs in the database server. However, there is no data lost in this scenario because the national character set is always a larger set than the database character set.

JDBC and SQLJ Programming with Unicode

Oracle provides three JDBC drivers for Java programs to access Unicode data in the database. They are the JDBC OCI driver, JDBC Thin driver, and JDBC KPRB driver. Java programs can insert or retrieve Unicode data to and from columns of SQL `CHAR` and `NCHAR` datatypes. Specifically, JDBC allows Java programs to bind or define Java strings to SQL `CHAR` and `NCHAR` datatypes. Because Java's `string` datatype is UTF-16 encoded, data retrieved from or inserted into the database must be converted from UTF-16 to the database character set or the national character set and vice versa. The SQLJ preprocessor allows Java programs to embed SQL statements to simplify database access code. It translates the embedded SQL statements of a Java program to the corresponding JDBC calls. Similar to JDBC, SQLJ allows programs to bind or define Java String to a SQL `CHAR` or `NCHAR` column. JDBC and SQLJ also allow you to specify the PL/SQL and SQL statements in Java strings so that any non-ASCII schema object names can be referenced in Java programs.

Java String Bind and Define in Unicode

Oracle JDBC drivers allow you to access SQL `CHAR` datatypes in the database using Java string bind or define variables. The following code illustrates how to bind or define a Java string to a `CHAR` column:

```
int empno = 12345;
String ename = "Joe"
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO" +
    "emp (ename, empno) VALUES (?, ?)");
pstmt.setString(1, ename);
pstmt.setInt(2, empno);
pstmt.execute();                /* execute to insert into first row */
empno += 1;                     /* next employee number */
ename = "\uFF2A\uFF4F\uFF45"; /* Unicode characters in name */
```

```
pstmt.setString(1, ename);
pstmt.setInt(2, empno);
pstmt.execute();                /* execute to insert into second row */
```

For binding or defining Java string variables to SQL NCHAR datatypes, Oracle extends the JDBC specification to add the `PreparedStatement.setFormOfUse()` method through which you can explicitly specify the target column of a bind variable to be a SQL NCHAR datatype. The following code illustrates how to bind a Java string to an NCHAR column:

```
int empno = 12345;
String ename = "Joe"
oracle.jdbc.OraclePreparedStatement pstmt =
    (oracle.jdbc.OraclePreparedStatement)
    conn.prepareStatement("INSERT INTO emp (ename, empno) VALUES (?, ?)");
pstmt.setFormOfUse(1, oracle.jdbc.OraclePreparedStatement.FORM_NCHAR);
pstmt.setString(1, ename);
pstmt.setInt(2, empno);
pstmt.execute();                /* execute to insert into first row */
empno += 1;                     /* next employee number */
ename = "\uFF2A\uFF4F\uFF45";  /* Unicode characters in name */
pstmt.setString(1, ename);
pstmt.setInt(2, empno);
pstmt.execute();                /* execute to insert into second row */
```

You can bind or define a Java string against an NCHAR column without explicitly specifying the form of use argument, but you then have the following implications:

1. If you do not specify the argument in the `setString()` method, JDBC assumes the bind or define variable to be for the SQL CHAR column. As a result, it tries to convert them to the database character set. When the data gets to the database, the database implicitly converts the data in the database character set to the national character set. During this conversion, data can be lost when the database character set is a subset of the national character set. Because the national character set is either UTF8 or AL16UTF16, data loss would happen if the database character set is not UTF8 or AL32UTF8.
2. Because implicit conversion from SQL CHAR to SQL NCHAR datatypes happens in the database, database performance will be adversely impacted.

In addition, if you bind or define a Java string for a column of SQL CHAR datatypes but specify the form of use argument, performance of the database will be adversely affected. However, data should not be lost because the national character set is always a larger set than the database character.

JDBC Restriction

You must place a `setFormOfUse()` statement before binding or defining Java variables to SQL `NCHAR` datatypes. The following code illustrates a sample setting of `setFormOfUse()`:

```
//-----
// Call dbms_lob.read(:clob, :read_this_time, :i+1, :string_this_time)
//-----
OracleCallableStatement cstmt = (oracle.jdbc.OracleCallableStatement)
    conn.prepareCall("BEGIN dbms_lob.read(:1, :2, :3, :4 ); END;");
while (i < length)
{
    cstmt.setFormOfUse(1,oracle.jdbc.Const.NCHAR);
    cstmt.setFormOfUse(4,oracle.jdbc.Const.NCHAR);

    cstmt.registerOutParameter(2,oracle.jdbc.OracleTypes.BIGINT);
    // **** the following 2 lines have to be put after setFormOfUse() ****
    cstmt.registerOutParameter(4,oracle.jdbc.OracleTypes.CHAR);
    cstmt.setCLOB(1,clob);
    cstmt.setLong(2,chunk);
    cstmt.setLong(3,i+1);
    cstmt.execute();
}
```

Java Data Conversion in Unicode

Because Java strings are always encoded in UTF-16, JDBC drivers transparently convert data from the database character set to UTF-16 or the national character set.

The conversion paths taken are different for the three JDBC drivers:

1. For the OCI driver, the SQL statements are always converted to the database character set by the driver before it is sent to the database for processing. For Java `string` bind or define variables, [Table 6-4](#) summarizes the conversion paths taken for different scenarios:

Table 6-4 OCI Driver Conversion Path

Form of Use	SQL Datatype	Conversion Path
Const.CHAR (Default)	CHAR	Java String to/from database character set happens in the JDBC driver

Table 6–4 OCI Driver Conversion Path (Cont.)

Form of Use	SQL Datatype	Conversion Path
Const.CHAR (Default)	NCHAR	Java String to/from database character set happens in the JDBC driver. Data in the database character set to/from national character set happens in the database server
Const.NCHAR	NCHAR	Java String to/from national character set happens in the JDBC driver
Const.NCHAR	CHAR	Java String to/from national character set happens in the JDBC driver Data in national character set to/from database character set happens in the database server

2. For the Thin driver, SQL statements are always converted to either the database character set or UTF-8 by the driver before they are sent to the database for processing. The Thin driver also notifies the database that a SQL statement requires further conversion before being processed. The database, in turn, converts the SQL statement to the database character set. For Java `string` bind and define variables, the conversion paths shown in Table 6–5 are taken for the Thin driver:

Table 6–5 Thin Driver Conversion Path

Form of Use	SQL Datatype	Database Character Set	Conversion Path
Const.CHAR (Default)	CHAR	US7ASCII or WE8ISO8859P1	Java String to/from the database character set happens in the Thin driver
Const.CHAR (Default)	NCHAR	US7ASCII or WE8ISO8859P1	Java String to/from the database character set happens in the Thin driver. Data in the database character set to/from the national character set happens in the database server
Const.CHAR (Default)	CHAR	non-ASCII and non-WE8ISO8859P1	Java String to/from UTF-8 happens in the Thin driver. Data in UTF-8 to/from the database character set happens in the database server

Table 6–5 Thin Driver Conversion Path (Cont.)

Form of Use	SQL Datatype	Database Character Set	Conversion Path
Const.CHAR (Default)	CHAR	non-ASCII and non-WE8ISO8859P1	Java String to/from UTF-8 happens in the Thin driver. Data in UTF-8 to/from national character set happens in the database server
Const.NCHAR	CHAR		Java String to/from the national character set happens in the Thin driver. Data in the national character set to/from the database character set happens in the database server
Const.NCHAR	NCHAR		Java String to/from the national character set happens in the Thin driver

3. The JDBC server-side internal driver is running in the server, all conversion are done in the database server. SQL statements specified as Java strings are converted to the database character set. Java `string` bind or define variables are converted to the database character sets if the form of use argument is not specified. Otherwise, they are converted to the national character set.

ODBC and OLEDB Programming with Unicode

You should use Oracle's ODBC and OLE DB drivers to access Oracle9i when using a Windows platform. This section describes how these drivers support Unicode.

Unicode-Enabled Drivers in ODBC and OLEDB

Oracle's ODBC and OLE DB drivers can handle Unicode data properly without data loss. For example, you can run a Unicode ODBC application containing Japanese data on English Windows if you install Japanese fonts and an input method editor for entering Japanese characters.

In Oracle9i, Oracle provides Windows platform-specific ODBC and OLE DB drivers only. For Unix platforms, contact your vendor.

OCI Dependency in Unicode

OCI Unicode binding and defining features are used by the ODBC and OLE DB drivers to handle Unicode data. As discussed in "[OCI Programming with Unicode](#)" on page 6-11, OCI Unicode data binding and defining features are independent

from NLS_LANG. This means Unicode data is handled properly, irrespective of the NLS_LANG setting on the platform.

ODBC and OLEDB Code Conversion in Unicode

In general, no redundant data conversion occurs unless you specify a different client datatype from that of the server. If you bind Unicode buffer SQL_C_WCHAR with a Unicode data column like NCHAR, for example, ODBC and OLE DB drivers bypass it between the application and OCI layer.

If you do not specify datatypes before fetching, and call SQLGetData with the client datatypes instead, the conversions in Table 6-6 occur:

Table 6-6 ODBC Implicit Binding Code Conversions

Datatypes of ODBC Client Buffer [*1]	Datatype of the Target Column for ODBC	Datatypes of the Target Column in the Database	Fetch Conversions	Comments
SQL_C_WCHAR	N/A	CHAR, VARCHAR2, CLOB	[*2] Database character set, NLS_LANG, to UTF-16 in OCI and ODBC [*3] Database character set to UTF-16 in OCI	No unexpected data loss, but may cause performance degradation if [*2]
SQL_C_CHAR	N/A	CHAR, VARCHAR2, CLOB	[*2] Database character set to NLS_LANG in OCI [*3] Database character set, UTF-16, to NLS_LANG character set in OCI and ODBC	No unexpected data loss, but may cause performance degradation in [*3] case

Note that you must specify the datatype for inserting and updating operations.

[*1] Datatype of ODBC client buffer is given when you call SQLGetData but not immediately. Hence, SQLFetch does not have the information.

[*2] If database character set is a subset of NLS_LANG.

[*3] If database character set is not a subset of NLS_LANG.

Because the ODBC driver guarantees data integrity, if you perform implicit bindings, redundant conversion may result in performance degradation. Your choice is the trade off between performance with explicit binding or usability with implicit binding.

OLE DB Code Conversions

Unlike ODBC, OLE DB only allows you to perform implicit bindings for both inserting/updating and fetching data. The conversion algorithm for determining the intermediate character set is the same as the implicit binding cases of ODBC.

Table 6–7 OLE DB Implicit Bindings

Datatypes of OLE DB Client Buffer	Datatypes of the Target Column in the Database	In- and Out-Binding Conversions	Comments
DBTYPE_WCHAR	CHAR, VARCHAR2, CLOB	[*1] Database character set to/from NLS_LANG character set in OCI. NLS_LANG character set to UTF-16 in OLE DB [*2] Database character set to/from UTF-16 in OCI	No unexpected data loss, but may cause performance degradation in [*2] case
DBTYPE_CHAR	CHAR, VARCHAR2, CLOB	[*1] Database character set to/from NLS_LANG in OCI [*2] Database character set to/from UTF-16 in OCI. UTF-16 to NLS_LANG character set in OLE DB	No unexpected data loss, but may cause performance degradation in [*3] case

[*1] If database character set is a subset of NLS_LANG.

[*2] If database character set is not a subset of NLS_LANG.

ODBC Unicode Datatypes

In ODBC Unicode applications, use `SQLWCHAR` to store Unicode data. All standard Windows Unicode functions can be used for `SQLWCHAR` data manipulations. For example, `wcslen` counts the number of characters of `SQLWCHAR` data:

```
SQLWCHAR sqlStmt[] = L"select ename from emp";
len = wcslen(sqlStmt);
```

Additionally, Microsoft's ODBC 3.5 specification defines three Unicode datatype identifiers for the `SQL_C_WCHAR`, `SQL_C_WVARCHAR`, and `SQL_WLONGVARCHAR` clients; and three Unicode datatype identifiers for servers `SQL_WCHAR`, `SQL_WVARCHAR`, and `SQL_WLONGVARCHAR`.

For binding operations, specify both datatypes for client and server using `SQLBindParameter`. The following is an example of Unicode binding, where the

client buffer Name indicates that Unicode data (SQL_C_WCHAR) is bound to the first bind variable associated with the Unicode column (SQL_WCHAR):

```
SQLBindParameter(StatementHandle, 1, SQL_PARAM_INPUT, SQL_C_WCHAR,
SQL_WCHAR, NameLen, 0, (SQLPOINTER)Name, 0, &Name);
```

To determine the ODBC Unicode datatypes for server, Table 6–8 represents the datatype mappings against SQL NCHAR datatypes:

Table 6–8 Server ODBC Unicode Datatype Mapping

ODBC Datatype	Oracle Datatype
SQL_WCHAR	NCHAR
SQL_WVARCHAR	NVARCHAR2
SQL_WLONGVARCHAR	NCLOB

According to ODBC specifications, SQL_WCHAR, SQL_WVARCHAR, and SQL_WLONGVARCHAR are treated as Unicode data, and are therefore measured in the number of characters instead of bytes to represents the data length when you retrieve table column information. Because NCHAR, NVARCHAR2, and NCLOB are migrated into pure Unicode datatypes, the above mappings will fit in the expected ODBC behavior.

OLEDB Unicode Datatypes

OLE DB offers you the choices of wchar_t *, BSTR, and OLESTR for the Unicode client C datatype. In practice, wchar_t is the most common datatype and the others are for specific purposes. The following example assigns a static SQL statement:

```
wchar_t *sqlStmt = OLESTR("SELECT ename FROM emp");
```

The OLESTR macro works exactly like an "L" modifier to indicate the Unicode string. If you need to allocate Unicode data buffer dynamically using OLESTR, use the IMalloc allocator (for example, CoTaskMemAlloc). However, using OLESTR is not the normal method for variable length data; use wchar_t* instead for generic string types. BSTR is similar but a string with a length prefix in the memory location preceding the string. Some functions and methods can accept only BSTR Unicode datatypes. Therefore, BSTR Unicode string must be manipulated with special functions like SysAllocString for allocation and SysFreeString for freeing memory.

Unlike ODBC, OLE DB does not allow you to specify the server datatype explicitly. When you set the client datatype, the OLE DB driver automatically performs data conversion if necessary.

Table 6–9 illustrates OLE DB datatype mapping:

Table 6–9 OLE DB Datatype Mapping

OLE DB Datatype	Oracle Datatype
DBTYPE_WCHAR	NCHAR or NVARCHAR2

If `DBTYPE_BSTR` is specified, it is assumed to be `DBTYPE_WCHAR` because both are Unicode strings.

ADO Access

ADO is a high-level API to access database via OLE DB and ODBC drivers. Most database application developers use the ADO interface on Windows because it is easily accessible from Visual Basic, the primary scripting language for Active Server Pages (ASP) for the Internet Information Server (IIS). To OLE DB and ODBC drivers, ADO is simply an OLE DB consumer or ODBC application. ADO assumes that OLE DB and ODBC drivers are Unicode-aware components; hence, it always attempts to manipulate Unicode data.

To use an ODBC driver with ADO, check the `Force SQL_WCHAR` attribute on the ODBC Data Source control panel. OLE DB is automatically adjusted to the ADO environment and requires no such action.

SQL Programming

This chapter contains information useful for SQL programming in a globalization support environment. It includes the following topics:

- [Locale-Dependent SQL Functions](#)
- [Time/Date/Calendar Formats](#)
- [Numeric Formats](#)
- [Miscellaneous Topics](#)

Locale-Dependent SQL Functions

All SQL functions whose behavior depends on NLS conventions allow NLS parameters to be specified. These functions are:

- TO_CHAR
- TO_DATE
- TO_NUMBER
- NLS_UPPER
- NLS_LOWER
- NLS_INITCAP
- NLSSORT

Explicitly specifying the optional NLS parameters for these functions allows the function evaluations to be independent of the NLS parameters in force for the session. This feature may be important for SQL statements that contain numbers and dates as string literals.

For example, the following query is evaluated correctly if the language specified for dates is AMERICAN:

```
SELECT ENAME FROM EMP
WHERE HIREDATE > '1-JAN-91';
```

Such a query can be made independent of the current date language by using these statements:

```
SELECT ENAME FROM EMP
WHERE HIREDATE > TO_DATE('1-JAN-91', 'DD-MON-YY',
    'NLS_DATE_LANGUAGE = AMERICAN');
```

In this way, language-independent SQL statements can be defined where necessary. For example, such statements might be necessary when string literals appear in SQL statements in views, CHECK constraints, or triggers.

All character functions support both single-byte and multibyte characters. Except where explicitly stated, character functions operate character by character, rather than byte by byte.

Default Specifications

When evaluating views and triggers, default values for NLS function parameters are taken from the values currently in force for the session. When evaluating `CHECK` constraints, default values are set by the NLS parameters that were specified at database creation.

Specifying Parameters

The syntax that specifies NLS parameters in SQL functions is:

```
'parameter = value'
```

The following NLS parameters can be specified:

- `NLS_DATE_LANGUAGE`
- `NLS_NUMERIC_CHARACTERS`
- `NLS_CURRENCY`
- `NLS_ISO_CURRENCY`
- `NLS_SORT`

Only certain NLS parameters are valid for particular SQL functions, as shown in [Table 7-1](#):

Table 7-1 SQL Functions and Their Parameters

SQL Function	Valid NLS Parameters
<code>TO_DATE</code>	<code>NLS_DATE_LANGUAGE</code> <code>NLS_CALENDAR</code>
<code>TO_NUMBER</code> :	<code>NLS_NUMERIC_CHARACTERS</code> <code>NLS_CURRENCY</code> <code>NLS_DUAL_CURRENCY</code> <code>NLS_ISO_CURRENCY</code>
<code>TO_CHAR</code>	<code>NLS_DATE_LANGUAGE</code> <code>NLS_NUMERIC_CHARACTERS</code> <code>NLS_CURRENCY</code> <code>NLS_ISO_CURRENCY</code> <code>NLS_DUAL_CURRENCY</code> <code>NLS_CALENDAR</code>

Table 7–1 SQL Functions and Their Parameters (Cont.)

SQL Function	Valid NLS Parameters
TO_NCHAR	NLS_DATE_LANGUAGE NLS_NUMERIC_CHARACTERS NLS_CURRENCY NLS_ISO_CURRENCY NLS_DUAL_CURRENCY NLS_CALENDAR
NLS_UPPER	NLS_SORT
NLS_LOWER	NLS_SORT
NLS_INITCAP	NLS_SORT
NLSSORT	NLS_SORT

Examples of the use of NLS parameters are:

```
TO_DATE ('1-JAN-89', 'DD-MON-YY',
        'nls_date_language = American')
```

```
TO_CHAR (hiredate, 'DD/MON/YYYY',
        'nls_date_language = French')
```

```
TO_NUMBER ('13.000,00', '99G999D99',
        'nls_numeric_characters = ','.'')
```

```
TO_CHAR (sal, '9G999D99L', 'nls_numeric_characters = ','.'
        nls_currency = ' Dfl'')
```

```
TO_CHAR (sal, '9G999D99C', 'nls_numeric_characters = ','.'
        nls_iso_currency = Japan')
```

```
NLS_UPPER (ename, 'nls_sort = Swiss')
```

```
NLSSORT (ename, 'nls_sort = German')
```

Note: For some languages, various lowercase characters correspond to a sequence of uppercase characters, or vice versa. As a result, the length of the output from the functions `NLS_UPPER`, `NLS_LOWER`, and `NLS_INITCAP` can differ from the input.

Unacceptable Parameters

Note that `NLS_LANGUAGE` and `NLS_TERRITORY` are not accepted as parameters in SQL functions, except for `NLSSORT`. Only NLS parameters that explicitly define the specific data items required for unambiguous interpretation of a format are accepted. `NLS_DATE_FORMAT` is also not accepted as a parameter for the reason described below.

If an NLS parameter is specified in `TO_CHAR`, `TO_NUMBER`, or `TO_DATE`, then a format mask must also be specified as the second parameter. For example, the following specification is legal:

```
TO_CHAR (hiredate, 'DD/MON/YYYY', 'nls_date_language = French')
```

The following specifications are illegal:

```
TO_CHAR (hiredate, 'nls_date_language = French')
```

```
TO_CHAR (hiredate, 'nls_date_language = French', 'DD/MON/YY')
```

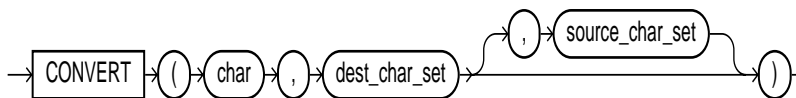
This restriction requires that a date format always be specified if an NLS parameter is in a `TO_CHAR` or `TO_DATE` function. As a result, `NLS_DATE_FORMAT` is not a valid NLS parameter for these functions.

CONVERT Function

The `CONVERT` SQL function allows conversion of character data between character sets.

The `CONVERT` function converts the binary representation of a character string in one character set to another. It uses exactly the same technique described previously for the conversion between database and client character sets. Hence, it uses replacement characters and has the same limitations.

The syntax for `CONVERT` is:

Figure 7–1 CONVERT Syntax

where `source_char_set` is the source character set and `dest_char_set` is the destination character set. If the `source_char_set` parameter is not specified, then it defaults to the database character set.

In client/server environments using different character sets, use the `TRANSLATE (...USING...)` statement to perform conversions instead of `CONVERT`. The conversion to client character sets will then properly know the server character set of the result of the `TRANSLATE` statement.

See Also:

- *Oracle9i SQL Reference* for more information about the `CONVERT` function
- ["Character Set Conversion Support"](#) on page A-18 for character set encodings that are used only for the `CONVERT` function

INSTR, LENGTH, and SUBSTR Functions and Character Sets

When using string manipulation functions, you can get different results depending upon the database character set. In particular, the `INSTR`, `LENGTH`, and `SUBSTR` functions can return incorrect results because of the difference between single and multibyte character sets. To guarantee correct results, you should use variants of these functions designed for multibyte character sets. These functions are variations of:

- [INSTR Functions and Character Sets](#)
- [LENGTH Functions and Character Sets](#)
- [SUBSTR Functions and Character Sets](#)

INSTR Functions and Character Sets

The instring functions search strings for the presence of a substring. In addition to the traditional `INSTR`, you can use `INSTRB`, `INSTR2`, `INSTR4`, and `INSTRC`. The functions allow you to return part of a string based on whether characters are calculated in bytes, UCS2 codepoints (where a surrogate pair is considered as two

codepoints), or UCS4 codepoints (treats a surrogate pair as one codepoint) or complete Unicode characters (same as `INSTR4` with additional support for treating composed Unicode characters as one code point).

The following examples highlights the differences between `INSTR` and `INSTRB` on a database where the database character set is UTF8.

This searches the string "Städte und Länder", beginning with the fifth character, for the character "d". It returns the character position in "Städte und Länder" at which the second occurrence of "d" begins.

```
SELECT INSTR('Städte und Länder','d', 5, 2) INSTR
FROM DUAL;
      INSTR
-----
      15
```

The following example searches the string "Städte und Länder", beginning with the fifth byte, for the character "d". It returns the byte position in "Städte und Länder" at which the second occurrence of "d" begins.

```
SELECT INSTRB('Städte und Länder','d', 5, 2) INSTRB
FROM DUAL;
      INSTRB
-----
      11
```

LENGTH Functions and Character Sets

The length functions return the length of a string. In addition to the traditional `LENGTH`, you can use `LENGTHB`, `LENGTH2`, `LENGTH4`, and `LENGTHC`. The functions allow you to return part of a string based on whether characters are calculated in bytes, UCS2 codepoints (where a surrogate pair is considered as two codepoints), or UCS4 codepoints (treats a surrogate pair as one codepoint) or complete Unicode characters (same as `LENGTH4` with additional support for treating composed Unicode characters as one code point).

The following examples highlight the differences between `LENGTH` and `LENGTHB` on a database where the database character set is UTF8.

```
SELECT LENGTH ('Télévision') LENGTH
FROM DUAL;
      LENGTH
-----
      10
```

```
SELECT LENGTHB ('Télévision') LENGTHB
FROM DUAL;
```

```
LENGTHB
-----
      12
```

SUBSTR Functions and Character Sets

The substring functions return the requested portion of a substring. In addition to the traditional SUBSTR, you can use SUBSTRB, SUBSTR2, SUBSTR4 and SUBSTRC. The functions allow you to return part of a string based on whether characters are calculated in bytes, UCS2 codepoints (where a surrogate pair is considered as two codepoints), or UCS4 codepoints (treats a surrogate pair as one codepoint) or complete Unicode characters (same as SUBSTR4 with additional support for treating composed Unicode characters as one code point).

The following examples highlight the differences between SUBSTR and SUBSTRB on a database where the database character set is AL32UTF8.

```
SELECT SUBSTR ('Fußball', 2 , 4) SUBSTR
FROM DUAL;
```

```
SUBS
----
ußba
```

```
SELECT SUBSTRB ('Fußball', 2 , 4) SUBSTRB
FROM DUAL;
```

```
SUB
---
ußb
```

LIKE Conditions and Character Sets

The LIKE conditions specify a test involving pattern matching. Whereas the equality operator (=) exactly matches one character value to another, the LIKE conditions match a portion of one character value to another by searching the first value for the pattern specified by the second. LIKE calculates strings using characters as defined by the input character set. LIKEC uses unicode complete characters. LIKE2 uses UCS2 codepoints. LIKE4 uses USC4 codepoints.

There is no LIKEB condition.

Character Set SQL Functions

Two SQL functions, `NLS_CHARSET_NAME` and `NLS_CHARSET_ID`, are provided to convert between character set ID numbers and character set names. They are used by programs that need to determine character set ID numbers for binding variables through OCI.

See Also: *Oracle9i SQL Reference*

Converting from Character Set Number to Character Set Name

The `NLS_CHARSET_NAME(n)` function returns the name of the character set corresponding to ID number *n*. The function returns `NULL` if *n* is not a recognized character set ID value.

Converting from Character Set Name to Character Set Number

`NLS_CHARSET_ID(TEXT)` returns the character set ID corresponding to the name specified by `TEXT`. `TEXT` is defined as a run-time `VARCHAR2` quantity, a character set name. Values for `TEXT` can be `NLSRTL` names that resolve to sets other than the database character set or the national character set.

If the value `CHAR_CS` is entered for `TEXT`, then the function returns the ID of the server's database character set. If the value `NCHAR_CS` is entered for `TEXT`, then the function returns the ID of the server's national character set. The function returns `NULL` if `TEXT` is not a recognized name. The value for `TEXT` must be entered in all uppercase.

Returning the Length of an NCHAR Column

`NLS_CHARSET_DECL_LEN(BYTECNT, CSID)` returns the declaration length (in number of characters) for an `NCHAR` column. The `BYTECNT` argument is the byte length of the column. The `CSID` argument is the character set ID of the column.

NLSSORT Function

The `NLSSORT` function replaces a character string with the equivalent sort string used by the linguistic sort mechanism. For a binary sort, the sort string is the same as the input string. The linguistic sort technique operates by replacing each character string with some other binary values, chosen so that sorting the resulting string produces the desired sorting sequence. When a linguistic sort is being used, `NLSSORT` returns the binary values that replace the original string.

The `ORDER BY` clause in a SQL statement is determined by the `NLS_SORT` session parameter, but it can be overridden by explicitly using the `NLSSORT` function, as the following example shows.

```
ALTER SESSION SET NLS_SORT = GERMAN;
SELECT *
FROM table1
ORDER BY col1;
```

The preceding example uses a German sort, but the following example uses a French one.

```
ALTER SESSION SET NLS_SORT = GERMAN;
SELECT *
FROM table1
ORDER BY NLSSORT(col1, 'NLS_SORT = FRENCH');
```

The `WHERE` clause normally uses binary comparison rather than linguistic comparison. But this can be overridden by the following methods.

- Use of the `NLSSORT` function in the `WHERE` clause.

```
SELECT *
FROM table1
WHERE NLSSORT(col1, 'NLS_SORT = FRENCH') >
      NLSSORT(col2, 'NLS_SORT = FRENCH');
```

- Setting the session parameter `NLS_COMP` to `ANSI`, in which case the `NLS_SORT` session parameter is used in the `WHERE` clause.

```
ALTER SESSION SET NLS_COMP = ANSI;
```

NLSSORT Syntax

There are four ways to use `NLSSORT`:

- `NLSSORT()`, which relies on the `NLS_SORT` parameter
- `NLSSORT(column1, 'NLS_SORT=xxxx')`
- `NLSSORT(column1, 'NLS_LANG=xxxx')`
- `NLSSORT(column1, 'NLS_LANGUAGE=xxxx')`

The `NLS_LANG` parameter of the `NLS_SORT` function is not the same as the `NLS_LANG` client environment setting. In the `NLSSORT` function, `NLS_LANG` specifies the abbreviated language name, for example, `US` for American or `PL` for Polish. An example is:

```
SELECT * FROM emp  
ORDER BY NLSSORT(col1, 'NLS_LANG=PL');
```

String Comparisons in a WHERE Clause

NLSSORT allows applications to perform string matching that follows alphabetic conventions. Normally, character strings in a WHERE clause are compared using the characters' binary values. A character is "greater than" another if it has a higher binary value in the database character set. Because the sequence of characters based on their binary values might not match the alphabetic sequence for a language, such comparisons often do not follow alphabetic conventions. For example, if a column (col1) contains the values ABC, ABZ, BCD, and ÄBC in the ISO 88591 8-bit character set, the following query:

```
SELECT col1 FROM tab1 WHERE col1 > 'B';
```

returns both BCD and ÄBC because Ä has a higher numeric value than B. However, in German, Ä is sorted alphabetically before B. Such conventions are language-dependent even when the same character is used. In Swedish, Ä is sorted after Z. Linguistic comparisons can be made using NLSSORT in the WHERE clause, as follows:

```
WHERE NLSSORT(col) comparison_operator NLSSORT(comparison_string)
```

Note that NLSSORT has to be on both sides of the comparison operator. For example:

```
SELECT col1 FROM tab1 WHERE NLSSORT(col1) > NLSSORT('B')
```

If a German linguistic sort is being used, this does not return strings beginning with Ä because, in the German alphabet, Ä comes before B. If a Swedish linguistic sort is being used, such names are returned because, in the Swedish alphabet, Ä comes after Z.

NLS_COMP

Normally, comparison in the WHERE clause or PL/SQL blocks is binary. You can use the NLSSORT function for linguistic comparison. Sometimes this can be tedious, especially when the linguistic sort needed has already been specified in the NLS_SORT session parameter. You can use NLS_COMP in such cases to indicate that the comparisons must be linguistic according to the NLS_SORT session parameter. This is done by altering the session:

```
ALTER SESSION SET NLS_COMP = ANSI;
```

To specify that comparison in the `WHERE` clause is always binary, issue the following statement:

```
ALTER SESSION SET NLS_COMP = BINARY;
```

When `NLS_COMP` is set to `ANSI`, a linguistic index improves the performance of the linguistic comparison. To enable a linguistic index, use the syntax:

```
CREATE INDEX i ON t(NLSSORT(col, 'NLS_SORT=FRENCH'));
```

Partitioned Tables and Indexes

String comparison for partition `VALUES LESS THAN` collation for DDL and DML always follows `BINARY` order.

Controlling an ORDER BY Clause

If a linguistic sorting sequence is in use, then `NLSSORT` is used implicitly on each character item in the `ORDER BY` clause. As a result, the sort mechanism (linguistic or binary) for an `ORDER BY` is transparent to the application. However, if the `NLSSORT` function is explicitly specified for a character item in an `ORDER BY` item, then the implicit `NLSSORT` is not done.

In other words, the `NLSSORT` linguistic replacement is only applied once, not twice. The `NLSSORT` function is generally not needed in an `ORDER BY` clause when the default sort mechanism is a linguistic sort. However, when the default sort mechanism is `BINARY`, then a query such as:

```
SELECT ename FROM emp  
ORDER BY ename
```

uses a binary sort. A German linguistic sort can be obtained using:

```
SELECT ename FROM emp  
ORDER BY NLSSORT(ename, 'NLS_SORT = GERMAN')
```

Time/Date/Calendar Formats

Several format masks are provided with the `TO_CHAR`, `TO_DATE`, and `TO_NUMBER` functions to format dates and numbers according to the relevant conventions.

Date Formats

A format element `RM` (Roman Month) returns a month as a Roman numeral. One can specify either uppercase or lowercase using `RM` or `rm` respectively. For

example, for the date 7 Sep 1998, `DD-rm-YYYY` will return `07-ix-1998` and `DD-RM-YYYY` will return `07-IX-1998`.

Note that the `MON` and `DY` format masks explicitly support month and day abbreviations that may not be three characters in length. For example, the abbreviations "Lu" and "Ma" can be specified for the French "Lundi" and "Mardi", respectively.

Week and Day Number Conventions

The week numbers returned by the `WW` format mask are calculated according to the algorithm `int((day-1jan1)/7)`. This week number algorithm does not follow the ISO standard (2015, 1992-06-15).

To support the ISO standard, a format element `IW` is provided that returns the ISO week number. In addition, format elements `I`, `IY`, `IYY`, and `IYYY`, equivalent in behavior to the format elements `Y`, `YY`, `YYY`, and `YYYY`, return the year relating to the ISO week number.

In the ISO standard, the year relating to an ISO week number can be different from the calendar year. For example, 1st Jan 1988 is in ISO week number 53 of 1987. A week always starts on a Monday and ends on a Sunday.

- If January 1 falls on a Friday, Saturday, or Sunday, then the week including January 1 is the last week of the previous year, because most of the days in the week belong to the previous year.
- If January 1 falls on a Monday, Tuesday, Wednesday, or Thursday, then the week is the first week of the new year, because most of the days in the week belong to the new year.

For example, January 1, 1991, is a Tuesday, so Monday, December 31, 1990, to Sunday, January 6, 1991, is week 1. Thus, the ISO week number and year for December 31, 1990, is 1, 1991. To get the ISO week number, use the format mask `IW` for the week number and one of the `IY` formats for the year.

Numeric Formats

Several additional format elements are provided for formatting numbers:

- `D` (decimal) returns the decimal point character.
- `G` (group) returns the group separator.
- `L` (local currency) returns the local currency symbol.

- `C` (international currency) returns the ISO currency symbol.
- `RN` (Roman numeral) returns the number as its Roman numeral equivalent.

For Roman numerals, one can specify either uppercase or lowercase, using `RN` or `rn`, respectively. The number being converted must be an integer in the range 1 to 3999.

See Also: *Oracle9i SQL Reference* for a complete description on using date and number masks

Miscellaneous Topics

The following topic is also important for SQL programming:

- [The Concatenation Operator](#)

The Concatenation Operator

If the database character set replaces the vertical bar `|` with a national character, then all SQL statements that use the concatenation operator (ASCII 124) will fail. For example, creating a procedure will fail because it generates a recursive SQL statement that uses concatenation. When you use a 7-bit replacement character set such as `D7DEC`, `F7DEC`, or `SF7ASCII` for the database character set, then the national character which replaces the vertical bar is not allowed in object names because the vertical bar is interpreted as the concatenation operator.

On the user side, one can use a 7-bit replacement character set if the database character set is the same or compatible, that is, if both character sets replace the vertical bar with the same national character.

OCI Programming

This chapter contains information useful for OCI programming, including:

- [Using the OCI NLS Functions](#)
- [NLS Language Information Retrieval](#)
- [String Manipulation in OCI](#)
- [Character Classification in OCI](#)
- [Character Set Conversion in OCI](#)
- [Messaging Mechanism in OCI](#)

Using the OCI NLS Functions

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

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

NLS Language Information Retrieval

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

Using environment handles, you can retrieve the following information:

- Days of the week (translated)
- Abbreviated days of the week (translated)
- Month names (translated)
- Abbreviated month names (translated)
- Yes/no (translated)
- AM/PM (translated)
- AD/BC (translated)
- Numeric format
- Debit/credit
- Date format
- Currency formats

- Default language
- Default territory
- Default character set
- Default linguistic sort
- Default calendar

OCINlsGetInfo

Syntax

```
sword OCINlsGetInfo(dvoid *hndl, OCIError *errhp, OraText *buf, size_t buflen,
ub2 item)
```

Remarks

This function generates language information specified by `item` from OCI environment or user session handle `hndl` into an array pointed to by `buf` within a size limitation as `buflen`. If the Unicode mode is enabled, then text data is returned in UTF-16.

See Also: [Chapter 6, "Unicode Programming"](#), for further details

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE, or OCI_ERROR on wrong items

Table 8–1 *OCINlsGetInfo Keywords/Parameters*

Keyword/ Parameter	Meaning
<code>hndl</code> (IN/SOUT)	The OCI environment or user session handle initialized in object mode
<code>errhp</code> (IN/OUT)	The OCI error handle. If there is an error, it is recorded in <code>errhp</code> and this function returns a NULL pointer. Diagnostic information can be obtained by calling <code>OCIErrorGet()</code>
<code>buf</code> (OUT)	Pointer to the destination buffer

Table 8–1 *OCI_NlsGetInfo Keywords/Parameters (Cont.)*

Keyword/ Parameter	Meaning
<code>buflen(IN)</code>	The size of the destination buffer. The maximum length for each piece of information is <code>OCI_NLS_MAXBUFSZ</code> bytes
<code>item(IN)</code>	Specifies which item in OCI environment handle to return. Can be one of the following values: <code>OCI_NLS_DAYNAME1</code> : Native name for Monday <code>OCI_NLS_DAYNAME2</code> : Native name for Tuesday <code>OCI_NLS_DAYNAME3</code> : Native name for Wednesday <code>OCI_NLS_DAYNAME4</code> : Native name for Thursday <code>OCI_NLS_DAYNAME5</code> : Native name for Friday <code>OCI_NLS_DAYNAME6</code> : Native name for Saturday <code>OCI_NLS_DAYNAME7</code> : Native name for Sunday <code>OCI_NLS_ABDAYNAME1</code> : Native abbreviated name for Monday <code>OCI_NLS_ABDAYNAME2</code> : Native abbreviated name for Tuesday <code>OCI_NLS_ABDAYNAME3</code> : Native abbreviated name for Wednesday <code>OCI_NLS_ABDAYNAME4</code> : Native abbreviated name for Thursday <code>OCI_NLS_ABDAYNAME5</code> : Native abbreviated name for Friday <code>OCI_NLS_ABDAYNAME6</code> : Native abbreviated name for Saturday <code>OCI_NLS_ABDAYNAME7</code> : Native abbreviated name for Sunday

Table 8–1 *OCINlsGetInfo Keywords/Parameters (Cont.)*

Keyword/ Parameter	Meaning
item (IN)	OCI_NLS_MONTHNAME1: Native name for January OCI_NLS_MONTHNAME2: Native name for February OCI_NLS_MONTHNAME3: Native name for March OCI_NLS_MONTHNAME4: Native name for April OCI_NLS_MONTHNAME5: Native name for May OCI_NLS_MONTHNAME6: Native name for June OCI_NLS_MONTHNAME7: Native name for July OCI_NLS_MONTHNAME8: Native name for August OCI_NLS_MONTHNAME9: Native name for September OCI_NLS_MONTHNAME10: Native name for October OCI_NLS_MONTHNAME11: Native name for November OCI_NLS_MONTHNAME12: Native name for December OCI_NLS_ABMONTHNAME1: Native abbreviated name for January OCI_NLS_ABMONTHNAME2: Native abbreviated name for February OCI_NLS_ABMONTHNAME3: Native abbreviated name for March OCI_NLS_ABMONTHNAME4: Native abbreviated name for April OCI_NLS_ABMONTHNAME5: Native abbreviated name for May OCI_NLS_ABMONTHNAME6: Native abbreviated name for June OCI_NLS_ABMONTHNAME7: Native abbreviated name for July OCI_NLS_ABMONTHNAME8: Native abbreviated name for August OCI_NLS_ABMONTHNAME9: Native abbreviated name for September OCI_NLS_ABMONTHNAME10: Native abbreviated name for October OCI_NLS_ABMONTHNAME11: Native abbreviated name for November OCI_NLS_ABMONTHNAME12: Native abbreviated name for December

Table 8–1 *OCI_NLS_GetInfo Keywords/Parameters (Cont.)*

Keyword/ Parameter	Meaning
item(IN)	OCI_NLS_YES: Native string for affirmative response
	OCI_NLS_NO: Native negative response
	OCI_NLS_AM: Native equivalent string of AM
	OCI_NLS_PM: Native equivalent string of PM
	OCI_NLS_AD: Native equivalent string of AD
	OCI_NLS_BC: Native equivalent string of BC
	OCI_NLS_DECIMAL: Decimal character
	OCI_NLS_GROUP: Group separator
	OCI_NLS_DEBIT: Native symbol of debit
	OCI_NLS_CREDIT: Native symbol of credit
	OCI_NLS_DATEFORMAT: Oracle date format
	OCI_NLS_INT_CURRENCY: International currency symbol
	OCI_NLS_DUAL_CURRENCY: Dual currency symbol
	OCI_NLS_LOC_CURRENCY: Locale currency symbol
	OCI_NLS_LANGUAGE: Language name
	OCI_NLS_ABLANGUAGE: Abbreviation for language name
	OCI_NLS_TERRITORY: Territory name
	OCI_NLS_CHARACTER_SET: Character set name
	OCI_NLS_LINGUISTIC_NAME: Linguistic name
	OCI_NLS_CALENDAR: Calendar name

Table 8–1 *OCINlsGetInfo Keywords/Parameters (Cont.)*

Keyword/ Parameter	Meaning
item(IN)	OCI_NLS_WRITING_DIR: Language writing direction
	OCI_NLS_ABTERITORY: Territory abbreviation
	OCI_NLS_DDATEFORMAT: Oracle default date format
	OCI_NLS_DTIMEFORMAT: Oracle default time format
	OCI_NLS_SFDATEFORMAT: Local string formatted date format
	OCI_NLS_SFTIMEFORMAT: Local string formatted time format
	OCI_NLS_NUMGROUPING: Number grouping fields
	OCI_NLS_LISTSEP: List separator
	OCI_NLS_MONDECIMAL: Monetary decimal character
	OCI_NLS_MONGROUP: Monetary group separator
	OCI_NLS_MONGROUPING: Monetary grouping fields
	OCI_NLS_INT_CURRENCYSEP: International currency separator

OCI_Nls_MaxBufSz

When calling `OCINlsGetInfo()`, you need to allocate the buffer to store the returned information for the particular language. The buffer size varies, depending on which item you are querying and what encoding you are using to store the information. Developers should not need to know how many bytes it takes to store "January" in Japanese using JA16SJIS encoding. That is exactly what `OCI_NLS_MAXBUFSZ` is used for; it guarantees that the `OCI_NLS_MAXBUFSZ` is big enough to hold the largest item returned by `OCINlsGetInfo()`. This guarantees that the largest item returned by `OCINlsGetInfo()` will fit in the buffer.

See Also:

- *Oracle Call Interface Programmer's Guide*
- *Oracle9i Data Cartridge Developer's Guide*

NLS Language Information Retrieval Sample Code

The following is a simple case of retrieving information and checking for errors.

```
sword MyPrintLinguisticName(envhp, errhp)
OCIEnv   *envhp;
OCIError *errhp;
```

```
{
    OraText  infoBuf[OCI-NLS_MAXBUFSZ];
    sword ret;

    ret = OCINlsGetInfo(envhp,                /* environment handle */
                        errhp,                /* error handle */
                        infoBuf,              /* destination buffer */
                        (size_t) OCI-NLS_MAXBUFSZ, /* buffer size */
                        (ub2) OCI-NLS_LINGUISTIC_NAME); /* item */

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

String Manipulation in OCI

Two types of data structure are supported for string manipulation: multibyte string and wide character string. Multibyte strings are in native Oracle character set encoding and functions operated on them take the string as a whole unit. Wide character string `wchar` functions provide more flexibility in string manipulation and support character-based and string-based operations.

The wide character datatype is Oracle-specific and not to be confused with the `wchar_t` defined by the ANSI/ISO C standard. The Oracle wide character is always 4 bytes in all platforms, while `wchar_t` is depends on the implementation and the platform. The idea of the Oracle wide character is to normalize multibyte character to have a fixed-width encoding for easy processing. This way, round-trip conversion between the Oracle wide character set and the native character set is guaranteed.

The string manipulation can be classified into the following categories:

- Conversion of string between multibyte and wide character
- Character classifications
- Case conversion

- Display length calculation
- General string manipulation, such as comparison, concatenation, and searching

Table 8–2 OCI String Manipulation Calls

Function Call	Description
<code>OCIMultiByteToWideChar()</code>	Converts an entire null-terminated string into the <code>wchar</code> format
<code>OCIMultiByteInSizeToWideChar()</code>	Converts part of a string into the <code>wchar</code> format
<code>OCIWideCharToMultiByte()</code>	Converts an entire null-terminated wide character string into a multibyte string
<code>OCIWideCharInSizeToMultiByte()</code>	Converts part of a wide character string into the multibyte format
<code>OCIWideCharToLower()</code>	If there is a upper-case character mapping in the specified locale, then it will return the lower-case in wide character. If not, it returns the same wide character.
<code>OCIWideCharToUpper()</code>	If there is an lower-case character mapping in the specified locale, it will return the upper-case in wide character. If not, it returns the same wide character.
<code>OCIWideCharStrcmp()</code>	Compares two wide character strings in binary, linguistic, or case-insensitive manners
<code>OCIWideCharStrncmp()</code>	Similar to <code>OCIWideCharStrcmp()</code> , but compares two multibyte strings in binary, linguistic, or case-insensitive manners, except that at most <code>len1</code> bytes form <code>str1</code> and <code>len2</code> bytes form <code>str2</code> are compared
<code>OCIWideCharStrcat()</code>	Appends a copy of the string pointed to by <code>wsrcstr</code> . Then it returns the number of characters in the resulting string.
<code>OCIWideCharStrchr()</code>	Searches for the first occurrence of <code>wc</code> in the string pointed to by <code>wstr</code> . Then returns a pointer to the <code>wchar</code> if successful
<code>OCIWideCharStrncpy()</code>	Copies the <code>wchar</code> string pointed to by <code>wsrcstr</code> into the array pointed to by <code>wdststr</code> . Then it returns the number of characters copied.
<code>OCIWideCharStrlen()</code>	Computes the number of characters in the <code>wchar</code> string pointed to by <code>wstr</code> , and returns this number
<code>OCIWideCharStrncat()</code>	Appends a copy of the string pointed to by <code>wsrcstr</code> . Then it returns the number of characters in the resulting string, except that at most <code>n</code> characters are appended.
<code>OCIWideCharStrncpy()</code>	Copies the <code>wchar</code> string pointed to by <code>wsrcstr</code> into the array pointed to by <code>wdststr</code> . Then it returns the number of characters copied, except that at most <code>n</code> characters are copied from the array.

Table 8–2 OCI String Manipulation Calls (Cont.)

Function Call	Description
<code>OCIWideCharStrrchr()</code>	Searches for the last occurrence of <code>wc</code> in the string pointed to by <code>wstr</code>
<code>OCIWideCharStrCaseConversion()</code>	Converts the wide character string pointed to by <code>wsrcstr</code> into case specified by a flag and copies the result into the array pointed to by <code>wdststr</code>
<code>OCIWideCharDisplayLength()</code>	Determines the number of column positions required for <code>wc</code> in display
<code>OCIWideCharMultibyteLength()</code>	Determines the number of bytes required for <code>wc</code> in multibyte encoding
<code>OCIMultiByteStrcmp()</code>	Compares two multibyte strings in binary, linguistic, or case-insensitive manners
<code>OCIMultiByteStrncmp()</code>	Compares two multibyte strings in binary, linguistic, or case-insensitive manners, except that at most <code>len1</code> bytes from <code>str1</code> and <code>len2</code> bytes from <code>str2</code> are compared
<code>OCIMultiByteStrcat()</code>	Appends a copy of the multibyte string pointed to by <code>srcstr</code>
<code>OCIMultiByteStrcpy()</code>	Copies the multibyte string pointed to by <code>srcstr</code> into an array pointed to by <code>dststr</code> . It returns the number of bytes copied
<code>OCIMultiByteStrlen()</code>	Computes the number of bytes in the multibyte string pointed to by <code>str</code> and returns this number
<code>OCIMultiByteStrncat()</code>	Appends a copy of the multibyte string pointed to by <code>srcstr</code> , except that at most <code>n</code> bytes from <code>srcstr</code> are appended to <code>dststr</code>
<code>OCIMultiByteStrncpy()</code>	Copies the multibyte string pointed to by <code>srcstr</code> into an array pointed to by <code>dststr</code> . It returns the number of bytes copied, except that at most <code>n</code> bytes are copied from the array pointed to by <code>srcstr</code> to the array pointed to by <code>dststr</code>
<code>OCIMultiByteStrnDisplayLength()</code>	Returns the number of display positions occupied by the complete characters within the range of <code>n</code> bytes
<code>OCIMultiByteStrCaseConversion()</code>	Converts part of a string from one character set to another

OCIMultiByteToWideChar

Syntax

```
sword OCIMultiByteToWideChar(dvoid *hndl, OCIWchar *dst, CONST OraText *src,
size_t *rsize);
```

Remarks

This routine converts an entire NULL-terminated string into the `wchar` format. The `wchar` output buffer will be NULL-terminated.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR

Table 8–3 OCIMultiByteToWideChar Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle to determine the character set of string
dst (OUT)	Destination buffer for wchar
src (IN)	Source string to be converted
rsize (OUT)	Number of characters converted including NULL-terminator. If it is a NULL pointer, nothing to return

OCIMultiByteInSizeToWideChar

Syntax

```
sword OCIMultiByteInSizeToWideChar(dvoid *hndl, OCIWchar *dst, size_t dstsz,
CONST OraText *src, size_t srcsz, size_t *rsize)
```

Remarks

This routine converts part of a string into the wchar format. It will convert as many complete characters as it can until it reaches the output buffer size or input buffer size or it reaches a NULL-terminator in source string. The output buffer will be NULL-terminated if space permits. If dstsz is zero, then this function will only return the number of characters not including the ending NULL terminator needed for converted string.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR

Table 8–4 OCIMultiByteInSizeToWideChar Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle to determine the character set of string
dst (OUT)	Pointer to a destination buffer for wchar. It can be NULL pointer when dstsz is zero
dstsz (IN)	Destination buffer size in character. If it is zero, this function just returns number of characters will be need for the conversion

Table 8–4 *OCIMultiByteInSizeToWideChar Keywords/Parameters (Cont.)*

Keyword/Parameter	Meaning
<code>src</code> (IN)	Source string to be converted
<code>srcsz</code> (IN)	Length of source string in bytes
<code>rsz</code> (OUT)	Number of characters written into destination buffer, or number of characters for converted string is <code>dstsz</code> is zero. If it is a <code>NULL</code> pointer, nothing to return

OCIWideCharToMultiByte

Syntax

```
sword OCIWideCharToMultiByte(dvoid *hndl, OraText *dst, CONST OCIWchar *src,
size_t *rsz)
```

Remarks

This routine converts an entire `NULL`-terminated wide character string into a multibyte string. The output buffer will be `NULL`-terminated.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE` or `OCI_ERROR`

Table 8–5 *OCIWideCharToMultiByte Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl</code> (IN/OUT)	OCI environment or user session handle to determine the character set of string
<code>dst</code> (OUT)	Destination buffer for multibyte string
<code>src</code> (IN)	Source <code>wchar</code> string to be converted
<code>srcsz</code> (IN)	Length of source string in bytes
<code>rsz</code> (OUT)	Number of characters written into destination buffer. If it is a <code>NULL</code> pointer, nothing will be returned

OCIWideCharInSizeToMultiByte

Syntax

```
sword OCIWideCharInSizeToMultiByte(dvoid *hndl, OraText *dst, size_t dstsz,
CONST OCIWchar *src, size_t srcsz, size_t *rsz)
```

Remarks

This routine converts part of `wchar` string into the multibyte format. It will convert as many complete characters as it can until it reaches the output buffer size, the input buffer size, or it reaches a `NULL`-terminator in source string. The output buffer will be `NULL`-terminated if space permits. If `dstsz` is zero, the function just returns the size of byte not including the ending `NULL`-terminator needed to store the converted string.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE` or `OCI_ERROR`

Table 8–6 OCIWideCharInSizeToMultiByte Keywords/Parameters

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set of string
<code>dst (OUT)</code>	Destination buffer for multibyte. It can be a <code>NULL</code> pointer if <code>dstsz</code> is zero
<code>dstsz (IN)</code>	Destination buffer size in bytes. If it is zero, it returns the size in bytes need for converted string
<code>src (IN)</code>	Source <code>wchar</code> string to be converted
<code>srcsz (IN)</code>	Length of source string in character
<code>rsz (OUT)</code>	Number of bytes written into destination buffer, or number of bytes need to store the converted string if <code>dstsz</code> is zero. If it is a <code>NULL</code> pointer, nothing to return

OCIWideCharToLower

Syntax

`OCIWchar OCIWideCharToLower(dvoid *hndl, OCIWchar wc)`

Remarks

If there is an upper-case character mapping for `wc` in the specified locale, it will return the lower-case in `wchar`, else return `wc` itself.

Returns

A `wchar`

Table 8–7 OCIWideCharToLower Keywords/Parameters

Keyword/Parameter	Meaning
hdl (IN/OUT)	OCI environment or user session handle to determine the character set
wc (IN)	wchar for upper-case mapping

OCIWideCharToUpper

Syntax

```
OCIWchar OCIWideCharToUpper(dvoid *hdl, OCIWchar wc)
```

Remarks

If there is a uppercase character mapping for `wc` in the specified locale, then it will return the uppercase character in `wchar`. Otherwise, it will return the `wc` itself.

Returns

A `wchar`

Table 8–8 OCIWideCharToUpper Keywords/Parameters

Keyword/Parameter	Meaning
hdl (IN/OUT)	OCI environment or user session handle to determine the character set
wc (IN)	wchar for uppercase mapping

OCIWideCharStrcmp

Syntax

```
int OCIWideCharStrcmp(dvoid *hdl, CONST OCIWchar *wstr1, CONST OCIWchar *wstr2,  
int flag)
```

Remarks

It compares two `wchar` strings in binary (based on `wchar` encoding value), linguistic, or case-insensitive.

Returns

- 0, if `wstr1 == wstr2`
- Positive, if `wstr1 > wstr2`
- Negative, if `wstr1 < wstr2`

Table 8–9 *OCIWideCharStrcmp Keywords/Parameters*

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle to determine the character set
wstr1 (IN)	Pointer to a NULL-terminated wchar string
wstr2 (IN)	Pointer to a NULL-terminated wchar string
flag (IN)	Is used to decide the comparison method. It can take one of the following values: OCI_NLS_BINARY: For the binary comparison, this is the default value. OCI_NLS_LINGUISTIC: For the linguistic comparison specified in the locale. This flag can be used with OR with OCI_NLS_CASE_INSENSITIVE for case-insensitive comparison

OCIWideCharStrncmp

Syntax

```
int OCIWideCharStrncmp(dvoid *hndl, CONST OCIWchar *wstr1, size_t len1, CONST OCIWchar *wstr2, size_t len2, int flag)
```

Remarks

This function is similar to OCIWideCharStrcmp(), except that at most len1 characters from wstr1 and len2 characters from wstr1 are compared. The NULL-terminator will be taken into the comparison.

Returns

- 0, if wstr1 = wstr2
- Positive, if wstr1 > wstr2
- Negative, if wstr1 < wstr2

Table 8–10 *OCIWideCharStrncmp Keywords/Parameters*

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle to determine the character set
wstr1 (IN)	Pointer to the first wchar string

Table 8–10 (Cont.) OCIWideCharStrncmp Keywords/Parameters (Cont.)

Keyword/Parameter	Meaning
len1 (IN)	The length for the first string for comparison
wstr2 (IN)	Pointer to the second wchar string
len2 (IN)	The length for the second string for comparison
flag (IN)	It is used to decide the comparison method. It can take one of the following values: OCI_NLS_BINARY: For the binary comparison, this is default value. OCI_NLS_LINGUISTIC: For the linguistic comparison specified in the locale. This flag can be used with OR with OCI_NLS_CASE_INSENSITIVE for case-insensitive comparison.

OCIWideCharStrcat

Syntax

```
size_t OCIWideCharStrcat(dvoid *hndl, OCIWchar *wdststr, CONST OCIWchar *wsrcstr)
```

Remarks

This function appends a copy of the wchar string pointed to by wsrcstr, including the NULL-terminator to the end of wchar string pointed to by wdststr.

Returns

The number of characters in the result string, not including the ending NULL-terminator.

Table 8–11 OCIWideCharStrcat Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle to determine the character set
wdststr (IN/OUT)	Pointer to the destination wchar string for appending
wsrcstr (IN)	Pointer to the source wchar string to append

OCIWideCharStrchr

Syntax

OCIWchar *OCIWideCharStrchr(dvoid *hndl, CONST OCIWchar *wstr, OCIWchar wc)

Remarks

This function searches for the first occurrence of `wc` in the `wchar` string pointed to by `wstr`.

Returns

A `wchar` pointer if successful, otherwise a `NULL` pointer

Table 8–12 OCIWideCharStrchr Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle to determine the character set
wstr (IN)	Pointer to the <code>wchar</code> string to search
wc (IN)	<code>wchar</code> to search for

OCIWideCharStrcpy

Syntax

size_t OCIWideCharStrcpy(dvoid *hndl, OCIWchar *wdststr, CONST OCIWchar *wsrctr)

Remarks

This function copies the `wchar` string pointed to by `wsrctr`, including the `NULL`-terminator, into the array pointed to by `wdststr`.

Returns

The number of characters copied not including the ending `NULL`-terminator.

Table 8–13 OCIWideCharStrcpy Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle to determine the character set
wdststr (OUT)	Pointer to the destination <code>wchar</code> buffer
wsrctr (IN)	Pointer to the source <code>wchar</code> string

OCIWideCharStrlen

Syntax

```
size_t OCIWideCharStrlen(dvoid *hndl, CONST OCIWchar *wstr)
```

Remarks

This function computes the number of characters in the `wchar` string pointed to by `wstr`, not including the `NULL`-terminator, and returns this number.

Returns

The number of characters not including ending `NULL`-terminator.

Table 8–14 *OCIWideCharStrlen Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl</code> (IN/OUT)	OCI environment or user session handle to determine the character set
<code>wstr</code> (IN)	Pointer to the source <code>wchar</code> string

OCIWideCharStrncat

Syntax

```
size_t OCIWideCharStrncat(dvoid *hndl, OCIWchar *wdststr, CONST OCIWchar *wsrctr, size_t n)
```

Remarks

This function is similar to `OCIWideCharStrcat()`, except that at most `n` characters from `wsrctr` are appended to `wdststr`. Note that the `NULL`-terminator in `wsrctr` will stop appending. `wdststr` will be `NULL`-terminated.

Returns

The number of characters in the result string, not including the ending `NULL`-terminator.

Table 8–15 *OCIWideCharStrncat Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl</code> (IN/OUT)	OCI environment or user session handle to determine the character set
<code>wdststr</code> (IN/OUT)	Pointer to the destination <code>wchar</code> string for appending
<code>wsrctr</code> (IN)	Pointer to the source <code>wchar</code> string to append

Table 8–15 *OCIWideCharStrncat Keywords/Parameters (Cont.)*

Keyword/Parameter	Meaning
<code>n (IN)</code>	Number of characters from <code>wsrcstr</code> to append

OCIWideCharStrncpy

Syntax

```
size_t OCIWideCharStrncpy(dvoid *hndl, OCIWchar *wdststr, CONST OCIWchar
*wsrcstr, size_t n)
```

Remarks

This function is similar to `OCIWideCharStrcpy()`, except that at most *n* characters are copied from the array pointed to by `wsrcstr` to the array pointed to by `wdststr`. Note that the NULL-terminator in `wdststr` will stop coping and result string will be NULL-terminated.

Returns

The number of characters copied not including the ending NULL-terminator.

Table 8–16 *OCIWideCharStrncpy Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wdststr (OUT)</code>	Pointer to the destination <code>wchar</code> buffer
<code>wsrcstr (IN)</code>	Pointer to the source <code>wchar</code> string
<code>n (IN)</code>	Number of characters from <code>wsrcstr</code> to copy

OCIWideCharStrrchr

Syntax

```
OCIWchar *OCIWideCharStrrchr(dvoid *hndl, CONST OCIWchar *wstr, OCIWchar wc)
```

Remarks

This function searches for the last occurrence of `wc` in the `wchar` string pointed to by `wstr`. It returns a pointer to the `wchar` if successful, or a NULL pointer.

Returns

`wchar` pointer if successful, otherwise a NULL pointer

Table 8–17 *OCIWideCharStrchr Keywords/Parameters*

Keyword/Parameter	Meaning
hdl (IN/OUT)	OCI environment or user session handle to determine the character set
wstr (IN)	Pointer to the wchar string to search
wc (IN)	wchar to search for

OCIWideCharStrCaseConversion

Syntax

```
size_t OCIWideCharStrCaseConversion(dvoid *hdl, OCIWchar *wdststr, CONST
OCIWchar*wsrctr, ub4 flag)
```

Remarks

This function converts the wide char string pointed to by `wsrctr` into the uppercase or lowercase specified by the flag and copies the result into the array pointed to by `wdststr`. The result string will be NULL-terminated.

Returns

The number of characters for the result string, not including NULL-terminator

Table 8–18 *OCIWideCharStrCaseConversion Keywords/Parameters*

Keyword/Parameter	Meaning
hdl (IN/OUT)	OCI environment or user session handle
wdststr (OUT)	Pointer to destination array
wsrctr (IN)	Pointer to source string
flag (IN)	Specify the case to convert: OCI_NLS_UPPERCASE: Convert to uppercase OCI_NLS_LOWERCASE: Convert to lowercase This flag can be used with OR with OCI_NLS_LINGUISTIC to specify that the linguistic setting in the locale will be used for case conversion.

OCIWideCharDisplayLength

Syntax

```
size_t OCIWideCharDisplayLength(dvoid *hndl, OCIWchar wc)
```

Remarks

This function determines the number of column positions required for `wc` in display. It returns the number of column positions, or 0 if `wc` is the NULL-terminator.

Returns

The number of display positions.

Table 8–19 OCIWideCharDisplayLength Keywords/Parameters

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wc (IN)</code>	wchar character

OCIWideCharMultiByteLength

Syntax

```
size_t OCIWideCharMultiByteLen(dvoid *hndl, OCIWchar wc)
```

Remarks

This function determines the number of byte required for `wc` in multibyte encoding. It returns the number of bytes in multibyte for `wc`.

Returns

The number of bytes.

Table 8–20 OCIWideCharMultiByteLength Keywords/Parameters

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wc (IN)</code>	wchar character

OCIMultiByteStrcmp

Syntax

```
int OCIMultiByteStrcmp(dvoid *hndl, CONST OraText *str1, CONST OraText *str2,
int flag)
```

Remarks

It compares two multibyte strings in binary (based on encoding value), linguistic, or case-insensitive.

Returns

- 0, if str1 == str2
- Positive, if str1 > str2
- Negative, if str1 < str2

Table 8–21 OCIMultiByteStrcmp Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle
str1 (IN)	Pointer to a NULL-terminated string
str2 (IN)	Pointer to a NULL-terminated string
flag (IN)	<p>It is used to decide the comparison method. It can take one of the following values:</p> <p>OCI_NLS_BINARY: For the binary comparison, this is the default value.</p> <p>OCI_NLS_LINGUISTIC: For the linguistic comparison specified in the locale.</p> <p>This flag can be used with OR with OCI_NLS_CASE_INSENSITIVE for case-insensitive comparison.</p>

OCIMultiByteStrncmp

Syntax

```
int OCIMultiByteStrncmp(dvoid *hndl, CONST OraText *str1, size_t len1, OraText
*str2, size_t len2, int flag)
```


Remarks

This function is similar to `OCIMultiByteStrcmp()`, except that at most `len1` bytes from `str1` and `len2` bytes from `str2` are compared. The `NULL`-terminator will be taken into the comparison.

Returns

- 0, if `str1 = str2`
- Positive, if `str1 > str2`
- Negative, if `str1 < str2`

Table 8–22 *OCIMultiByteStrncmp Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle
<code>str1 (IN)</code>	Pointer to the first string
<code>len1 (IN)</code>	The length for the first string for comparison
<code>str2 (IN)</code>	Pointer to the second string
<code>len2 (IN)</code>	The length for the second string for comparison
<code>flag (IN)</code>	It is used to decide the comparison method. It can take one of the following values: OCI_NLS_BINARY: For the binary comparison, this is the default value. OCI_NLS_LINGUISTIC: For the linguistic comparison specified in the locale. This flag can be used with OR with OCI_NLS_CASE_INSENSITIVE for case-insensitive comparison.

OCIMultiByteStrcat

Syntax

`size_t OCIMultiByteStrcat(dvoid *hndl, OraText *dststr, CONST OraText *srcstr)`

Remarks

This function appends a copy of the multibyte string pointed to by `srcstr`, including the `NULL`-terminator to the end of string pointed to by `dststr`. It returns the number of bytes in the result string not including the ending `NULL`-terminator.

Returns

The number of bytes in the result string, not including the ending NULL-terminator

Table 8–23 OCIMultiByteStrcat Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle to determine the character set
dststr (IN/OUT)	Pointer to the destination multibyte string for appending
srcstr (IN)	Pointer to the source string to append

OCIMultiByteStrcpy

Syntax

size_t OCIMultiByteStrcpy(dvoid *hndl, OraText *dststr, CONST OraText *srcstr)

Remarks

This function copies the multibyte string pointed to by `srcstr`, including the NULL-terminator, into the array pointed to by `dststr`. It returns the number of bytes copied, not including the ending NULL-terminator.

Returns

The number of bytes copied, not including the ending NULL-terminator

Table 8–24 OCIMultiByteStrcpy Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	Pointer to the OCI environment or user session handle
dststr (OUT)	Pointer to the destination buffer
srcstr (IN)	Pointer to the source multibyte string

OCIMultiByteStrlen

Syntax

size_t OCIMultiByteStrlen(dvoid *hndl, CONST OraText *str)

Remarks

This function computes the number of bytes in the multibyte string pointed to by `str`, not including the NULL-terminator, and returns this number.

Returns

The number of bytes not including ending `NULL`-terminator

Table 8–25 *OCIMultiByteStrlen Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	Pointer to the OCI environment or user session handle
<code>str (IN)</code>	Pointer to the source multibyte string

OCIMultiByteStrncat

Syntax

```
size_t OCIMultiByteStrncat(dvoid *hndl, OraText *dststr, CONST OraText *srcstr,
size_t n)
```

Remarks

This function is similar to `OCIMultiByteStrcat()`, except that at most *n* bytes from `srcstr` are appended to `dststr`. Note that the `NULL`-terminator in `srcstr` will stop appending and the function will append as many character as possible within *n* bytes. `dststr` will be `NULL`-terminated.

Returns

The number of bytes in the result string, not including the ending `NULL`-terminator.

Table 8–26 *OCIMultiByteStrncat Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	Pointer to OCI environment or user session handle
<code>dststr (IN/OUT)</code>	Pointer to the destination multibyte string for appending
<code>srcstr (IN)</code>	Pointer to the source multibyte string to append
<code>n (IN)</code>	The number of bytes from <code>srcstr</code> to append

OCIMultiByteStrncpy

Syntax

```
size_t OCIMultiByteStrncpy(dvoid *hndl, OraText *dststr, CONST OraText *srcstr,
size_t n)
```

Remarks

This function is similar to `OCIMultiByteStrncpy()`, except that at most *n* bytes are copied from the array pointed to by `srcstr` to the array pointed to by `dststr`. Note that the NULL-terminator in `srcstr` will stop coping and the function will copy as many character as possible within *n* bytes. The result string will be NULL-terminated.

Returns

The number of bytes copied not including the ending NULL-terminator

Table 8–27 *OCIMultiByteStrncpy Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	Pointer to OCI environment or user session handle
<code>srcstr (OUT)</code>	Pointer to the destination buffer
<code>dststr (IN)</code>	Pointer to the source multibyte string
<code>n (IN)</code>	The number of bytes from <code>srcstr</code> to copy

OCIMultiByteStrnDisplayLength

Syntax

`size_t OCIMultiByteStrnDisplayLength(dvoid *hndl, CONST OraText *str1, size_t n)`

Remarks

This function returns the number of display positions occupied by the complete characters within the range of *n* bytes.

Returns

The number of display positions.

Table 8–28 *OCIMultiByteStrncpy Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle
<code>str (IN)</code>	Pointer to a multibyte string
<code>n (IN)</code>	The number of bytes to examine

OCIMultiByteStrCaseConversion

Syntax

```
size_t OCIMultiByteStrCaseConversion(dvoid *hndl, OraText *dststr, CONST OraText
*srcstr, ub4 flag)
```

Remarks

This function convert the multibyte string pointed to by `srcstr` into the uppercase or lowercase specified by `flag` and copies the result into the array pointed to by `dststr`. The result string will be NULL-terminated.

Returns

The number of bytes for result string, not including NULL-terminator

Table 8–29 OCIMultibyteStrCaseConversion Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	OCI environment or user session handle
dststr (OUT)	Pointer to destination array
srcstr (IN)	Pointer to source string
flag (IN)	Specify the case to convert: OCI_NLS_UPPERCASE: Convert to uppercase OCI_NLS_LOWERCASE: Convert to lowercase This flag can be used with OR with OCI_NLS_LINGUISTIC to specify that the linguistic setting in the locale will be used for case conversion

String Manipulation Sample Code

The following is a simple case of handling string manipulation.

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

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

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

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

See Also:

- *Oracle Call Interface Programmer's Guide*
- *Oracle9i Data Cartridge Developer's Guide*

Character Classification in OCI

The Oracle Call Interface offers many function calls for classifying characters.

Table 8–30 OCI Character Classification Calls

Function Call	Description
OCIWideCharIsAlnum()	Tests whether the wide character is a letter or decimal digit
OCIWideCharIsAlpha()	Tests whether the wide character is an alphabetic letter
OCIWideCharIsCntrl()	Tests whether the wide character is a control character
OCIWideCharIsDigit()	Tests whether the wide character is a decimal digital character
OCIWideCharIsGraph()	Tests whether the wide character is a graph character
OCIWideCharIsLower()	Tests whether the wide character is a lowercase letter
OCIWideCharIsPrint()	Tests whether the wide character is a printable character
OCIWideCharIsPunct()	Tests whether the wide character is a punctuation character
OCIWideCharIsSpace()	Tests whether the wide character is a space character

Table 8–30 OCI Character Classification Calls (Cont.)

Function Call	Description
<code>OCIWideCharIsUpper()</code>	Tests whether the wide character is an uppercase character
<code>OCIWideCharIsXdigit()</code>	Tests whether the wide character is a hexadecimal digit
<code>OCIWideCharIsSingleByte()</code>	Tests whether <code>wc</code> is a single-byte character when converted into multibyte

OCIWideCharIsAlnum

Syntax

```
boolean OCIWideCharIsAlnum(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether `wc` is a letter or decimal digit.

Returns

TRUE or FALSE.

Table 8–31 OCIWideCharIsAlnum Keywords/Parameters

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wc (IN)</code>	wchar for testing

OCIWideCharIsAlpha

Syntax

```
boolean OCIWideCharIsAlpha(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether `wc` is an alphabetic letter.

Returns

TRUE or FALSE.

Table 8–32 OCIWideCharIsAlpha Keywords/Parameters

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set

Table 8–32 *OCIWideCharIsAlpha Keywords/Parameters (Cont.)*

Keyword/Parameter	Meaning
<code>wc (IN)</code>	<code>wchar</code> for testing

OCIWideCharIsCntrl

Syntax

```
boolean OCIWideCharIsCntrl(dvoid *hdl, OCIWchar wc)
```

Remarks

It tests whether `wc` is a control character.

Returns

TRUE or FALSE.

Table 8–33 *OCIWideCharIsCntrl Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hdl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wc (IN)</code>	<code>wchar</code> for testing

OCIWideCharIsDigit

Syntax

```
boolean OCIWideCharIsDigit(dvoid *hdl, OCIWchar wc)
```

Remarks

It tests whether `wc` is a decimal digit character.

Returns

TRUE or FALSE.

Table 8–34 *OCIWideCharIsDigit Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hdl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wc (IN)</code>	<code>wchar</code> for testing

OCIWideCharIsGraph

Syntax

```
boolean OCIWideCharIsGraph(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether `wc` is a graph character. A graph character is character with a visible representation and normally includes an alphabetic letter, decimal digit, and punctuation.

Returns

TRUE or FALSE

Table 8–35 *OCIWideCharIsGraph Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wc (IN)</code>	<code>wchar</code> for testing

OCIWideCharIsLower

Syntax

```
boolean OCIWideCharIsLower(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether `wc` is a lowercase letter.

Returns

TRUE or FALSE

Table 8–36 *OCIWideCharIsLower Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wc (IN)</code>	<code>wchar</code> for testing

OCIWideCharIsPrint

Syntax

```
boolean OCIWideCharIsPrint(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether `wc` is a printable character.

Returns

TRUE or FALSE

Table 8–37 *OCIWideCharIsPrint Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl</code> (IN/OUT)	OCI environment or user session handle to determine the character set
<code>wc</code> (IN)	<code>wchar</code> for testing

OCIWideCharIsPunct

Syntax

```
boolean OCIWideCharIsPunct(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether `wc` is a punctuation character.

Returns

TRUE or FALSE

Table 8–38 *OCIWideCharIsPunct Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl</code> (IN/OUT)	OCI environment or user session handle to determine the character set
<code>wc</code> (IN)	<code>wchar</code> for testing

OCIWideCharIsSpace

Syntax

```
boolean OCIWideCharIsSpace(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether `wc` is a space character. A space character only causes white space in displayed text (for example, space, tab, carriage return, newline, vertical tab or form feed).

Returns

TRUE or FALSE

Table 8–39 *OCIWideCharIsSpace Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wc (IN)</code>	<code>wchar</code> for testing

OCIWideCharIsUpper

Syntax

```
boolean OCIWideCharIsUpper(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether `wc` is an uppercase letter.

Returns

TRUE or FALSE

Table 8–40 *OCIWideCharIsUpper Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	OCI environment or user session handle to determine the character set
<code>wc (IN)</code>	<code>wchar</code> for testing

OCIWideCharIsXdigit

Syntax

```
boolean OCIWideCharIsXdigit(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether `wc` is a hexadecimal digit (0-9, A-F, a-f).

Returns

TRUE or FALSE

Table 8–41 OCIWideCharIsXdigit Keywords/Parameters

Keyword/Parameter	Meaning
hdl (IN/OUT)	OCI environment or user session handle to determine the character set
wc (IN)	wchar for testing

OCIWideCharIsSingleByte

Syntax

boolean OCIWideCharIsSingleByte(dvoid *hdl, OCIWchar wc)

Remarks

It tests whether wc is a single-byte character when converted into multibyte.

Returns

TRUE or FALSE

Table 8–42 OCIWideCharIsSingleByte Keywords/Parameters

Keyword/Parameter	Meaning
hdl (IN/OUT)	OCI environment or user session handle to determine the character set
wc (IN)	wchar for testing

Example 8–1 Character Classification Sample Code

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

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

```

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

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

return(status);
}

```

See Also:

- *Oracle Call Interface Programmer's Guide*
- *Oracle9i Data Cartridge Developer's Guide*

Character Set Conversion in OCI

Conversion between Oracle character set and Unicode (16-bit, fixed-width Unicode encoding) is supported. Replacement characters are used if there a character has no mapping from Unicode to the Oracle character set. Therefore, conversion back to the original character set is not always possible.

Table 8–43 OCI Character Set Conversion Calls

Function Call	Description
OCICharsetToUnicode()	Converts a multibyte string pointed to by <i>src</i> to Unicode into the array pointed to by <i>dst</i>
OCIUnicodeToCharset()	Converts a Unicode string pointed to by <i>src</i> to multibyte into the array pointed to by <i>dst</i>
OCICharSetConversionIsReplacementUsed()	Indicates whether the replacement character was used for nonconvertible characters in character set conversion in the last invocation of OCICharsetConv()

OCICharSetToUnicode

Syntax

```
sword OCICharSetToUnicode(dvoid *hndl, ub2 *dst, size_t dstlen, CONST OraText
*src, size_t srclen, size_t *rsize)
```

Remarks

This function converts a multibyte string pointed to by `src` to Unicode into the array pointed to by `dst`. The conversion will stop when it reaches the source limitation or destination limitation. The function will return the number of characters converted into Unicode. If `dstlen` is zero, it will just return the number of characters into `rsize` for the result without real conversion.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR

Table 8–44 OCICharSetToUnicode Keywords/Parameters

Keyword/Parameter	Meaning
hndl (IN/OUT)	Pointer to an OCI environment or user session handle
dst (OUT)	Pointer to a destination buffer
dstlen (IN)	The size of the destination buffer in characters
src (IN)	Pointer to multibyte source string
srclen (IN)	The size of source string in bytes
rsize (OUT)	The number of characters converted. If it is a NULL pointer, nothing to return

OCIUnicodeToCharSet

Syntax

```
sword OCIUnicodeToCharSet(dvoid *hndl, OraText *dst, size_t dstlen, CONST ub2
*src, size_t srclen, size_t *rsize)
```

Remarks

This function converts a Unicode string pointed to by `src` to multibyte into the array pointed to by `dst`. The conversion will stop when it reach to the source limitation or destination limitation. The function will return the number of bytes converted into multibyte. If `dstlen` is zero, it will just return the number of bytes into `rsize` for the result without real conversion.

If a Unicode character is not convertible for the character set specified in OCI environment or user session handle, a replacement character will be used for it. In this case, `OCICharsetConversionIsReplacementUsed()` will return true.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE` or `OCI_ERROR`

Table 8–45 *OCIUnicodeToCharSet Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	Pointer to an OCI environment or user session handle
<code>dst (OUT)</code>	Pointer to a destination buffer
<code>dstlen (IN)</code>	The size of destination buffer in bytes
<code>src (IN)</code>	Pointer to a Unicode string
<code>srclen (IN)</code>	The size of the source string in characters
<code>rsize (OUT)</code>	The number of bytes converted. If it is a <code>NULL</code> pointer, nothing to return

OCICharsetConversionIsReplacementUsed

Syntax

```
boolean OCICharsetConversionIsReplacementUsed(dvoid *hndl)
```

Remarks

This function indicates whether the replacement character was used for nonconvertible characters in character set conversion in the last invocation of `OCICharsetToUnicode()`.

Returns

`TRUE` if the replacement character was used when `OCICharsetConv()` was last invoked, else `FALSE`.

Table 8–46 *OCICharsetConversionIsReplacementUsed Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hndl (IN/OUT)</code>	Pointer to an OCI environment or user session handle

Conversion between the Oracle character set and Unicode (16-bit, fixed-width Unicode encoding) is supported. Replacement characters are used if there is no

mapping for a character from Unicode to the Oracle character set. thus, conversion back to the original character set is not always possible.

Example 8–2 Character Set Conversion Sample Code

The following is a simple conversion into Unicode:

```
size_t MyConvertMultiByteToUnicode(envhp, dstBuf, dstSize, srcStr)
OCIEnv *envhp;
ub2     *dstBuf;
size_t  dstSize;
OraText *srcStr;
{
    sword ret;
    size_t dstLen = 0;
    size_t srcLen;

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

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

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


See Also:

- *Oracle Call Interface Programmer's Guide*
- *Oracle9i Data Cartridge Developer's Guide*

Messaging Mechanism in OCI

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

Table 8–47 OCI Messaging Function Calls

Function Call	Description
OCIMessageOpen()	Opens a message handle for facility of product in a language pointed to by <code>hndl</code> .
OCIMessageGet()	Retrieves a message with message number identified by <code>msgno</code> . If the buffer is not zero, then the function will copy the message into the buffer pointed to by <code>msgbuf</code> .
OCIMessageClose ()	Closes a message handle pointed to by <code>msggh</code> and frees any memory associated with this handle.

See Also:

- *Oracle Call Interface Programmer's Guide*
- *Oracle9i Data Cartridge Developer's Guide*

OCIMessageOpen

Syntax

```
sword OCIMessageOpen(dvoid *hndl, OCIError *errhp, OCIMsg **msgghp, CONST OraText
*product, CONST OraText *facility, OCIDuration dur)
```

Remarks

This function opens a message-handling facility in a language pointed to by `hndl`. It first tries to open the message file corresponding to `hndl` for the facility. If it succeeds, it will use that file to initialize a message handle. Otherwise it will use the default message file, whose language is `AMERICAN`. The function returns a pointer pointed to a message handle into the `msgghp` parameter. If the Unicode mode is enabled, then the text parameters must be in UTF-16.

See Also: [Chapter 6, "Unicode Programming"](#)

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR

Table 8–48 *OCICharSetConversionIsReplacementUsed Keywords/Parameters*

Keyword/Parameter	Meaning
hndl (IN/OUT)	Pointer to an OCI environment or user session handle for message language
errhp (IN/OUT)	The OCI error handle. If there is an error, it is recorded in errhp and this function returns a NULL pointer. Diagnostic information can be obtained by calling OCIErrorGet ().
msghp (OUT)	A message handle for return
product (IN)	A pointer to a product name. Product name is used to locate the directory for messages depending on the operating system. For example, in Solaris, the directory of message files for the rdbms product is \${ORACLE_HOME}/rdbms.
facility (IN)	A pointer to a facility name in the product. It is used to construct a message file name. A message file name follows the conversion with facility as prefix. For example, the message file name for the img facility in the American language will be imgus.msb, where us is the abbreviation for the American language and msb is the message binary file extension.
dur (IN)	The duration for memory allocation for the return message handle. It can be the following values: OCI_DURATION_PROCESS OCI_DURATION_STATEMENT OCI_DURATION_SESSION

OCIMessageGet

Syntax

OraText *OCIMessageGet(OCIMsg *msggh, ub4 msgno, OraText *msgbuf, size_t buflen)

Remarks

This function will get a message with the message number identified by msgno. If buflen is not zero, then the function will copy the message into the buffer pointed to by msgbuf. If buflen is zero, then the message will be copied into a message

buffer inside the message handle pointed to by `msg_h`. For both cases, it will return the pointer to the NULL-terminated message string. If it cannot get the requested message, then it will return a NULL pointer. If the Unicode mode is enabled, then the message is returned in UTF-16.

See Also: [Chapter 6, "Unicode Programming"](#), for further details

Returns

If successful, a pointer to a NULL-terminated message string on success. Otherwise a NULL pointer

Table 8–49 *OCIMessageGet Keywords/Parameters*

Keyword/Parameter	Meaning
<code>msg_h (IN/OUT)</code>	Pointer to a message handle which was previously opened by <code>OCIMessageOpen ()</code>
<code>msgno (IN)</code>	The message number for getting message
<code>msgbuf (OUT)</code>	Pointer to a destination buffer to the message retrieved. If <code>buflen</code> is zero, it can be a NULL pointer
<code>buflen (IN)</code>	The size of the above destination buffer

OCIMessageClose

Syntax

```
sword OCIMessageClose(dvoid *hdl, OCIError *errhp, OCIMsg *msg_h)
```

Remarks

This function closes a message handle pointed to by `msg_h` and frees any memory associated with this handle.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR

Table 8–50 *OCIMessageClose Keywords/Parameters*

Keyword/Parameter	Meaning
<code>hdl (IN/OUT)</code>	Pointer to an OCI environment or user session handle for message language

Table 8–50 OCIMessageClose Keywords/Parameters (Cont.)

Keyword/Parameter	Meaning
errhp (IN/OUT)	The OCI error handle. If there is an error, it is record in errhp and this function returns a NULL pointer. Diagnostic information can be obtained by calling OCLErrorGet ()
msgsh (IN/OUT)	A pointer to a message handle that was previously opened by OCIMessageOpen ()

LMSGEN

Remarks

The lmsgen utility converts text-based message files (.msg) into binary format (.msb).

Syntax

LMSGEN *text_file* *product* *facility* [*language*]

where

text_file is a message text file
product is the name of the product
facility is the name of the facility
[*language*] is the optional message language in
<language>_<territory>.<character set> format

This is required if the message file is not tagged properly with language.

Text Message File Format

- Lines that start with "/" and "/" are treated as internal comments and are ignored.
- To tag the message file with a specific language:
CHARACTER_SET_NAME= Japanese_Japan.JA16EUC
- Each message is composed of 3 fields:
<message #>, <warning level #>, <message text>
 - Message # has to be unique within a message file.
 - Warning level # is not currently use. Use 0.

- Message text cannot be longer than 76 bytes.

Example

```
/ Copyright (c) 1988 by the Oracle Corporation. All rights reserved.
/ This is a testing us7ascii message file
# CHARACTER_SET_NAME= american_america.us7ascii
/
00000, 00000, "Export terminated unsuccessfully\n"
00003, 00000, "no storage definition found for segment(%lu, %lu)"
```

Message Example

Settings

This example retrieves messages from a `.msb` message file. The following settings are used:

```
product = $HOME/myApp
facility = imp
Language = American language
```

Based on the above setting, the `$HOME/myApp/mesg/impus.msb` message file will be used.

Message file

`Lmsgen` converts the message file (`impus.msg`) into binary format (`impus.msb`).

The following is a portion of the text message file called `impus.msg`:

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

Example 8-3 Messaging Sample Code

```
/* Assume that the OCI environment or user session handle, product, facility and
cache size are all initialized properly. */
...
OCIMsg msghnd; /* message handle */
/* initialize a message handle for retrieving messages from impus.msg */
err = OCIMessageOpen(hndl, errhp, &msghnd, prod, fac, OCI_DURATION_SESSION);
if (err != OCI_SUCCESS)
/* error handling */
```

```
...
                                /* retrieve the message with message number = 128 */
msgptr = OCIMessageGet(msghnd, 128, msgbuf, sizeof(msgbuf));
                                /* do something with the message, such as display it */
...
                                /* close the message handle when we has no more message to retrieve */
OCIMessageClose(hndl, errhp, msghnd);
```

Java Programming

This chapter examines globalization support for individual Java components. It includes the following topics:

- Overview of Oracle9i Java Support
- JDBC
- SQLJ
- Java Virtual Machine
- Java Stored Procedures
- Java Servlets and Java Server Pages
- CORBA and EJB
- Configurations for Multilingual Applications
- Multilingual Demo Applications in SQLJ

Overview of Oracle9i Java Support

Java support is included in all tiers of a multitier computing environment so that you can develop and deploy Java programs. You can run Java classes as Java stored procedures, Java servlets, Java CORBA objects, and Enterprise Java Beans (EJB) on the Java Virtual Machine (Oracle JVM) of the Oracle9i database. You can develop a Java class, load it into the database, and package it as a stored procedure that can be called from SQL. You can develop a Java servlet, load it on the database, and publish it as a callable servlet from a web browser. You can also develop a standard Java CORBA object or EJB, load the related classes into the database and publish them as named objects that are callable from any CORBA or EJB client.

The JDBC driver and SQLJ translator are also provided as programmatic interfaces that enable Java programs to access the Oracle9i database. You can write a Java application using JDBC or SQLJ programs with embedded SQL statements to access the database. Globalization support is provided across all these Java components to ensure that they function properly across databases of different character sets and language environments, and that they enable the development and deployment of multilingual Java applications for Oracle9i.

This chapter examines globalization support for individual Java components. Typical database and client configurations for multilingual application deployment are discussed, including an explanation of how the Java components are used in the configurations. Finally, the design and implementation of a sample application are explored to demonstrate how Oracle's Java support is used to make the application run in a multilingual environment.

Java components provide globalization support and use Unicode as the multilingual character. The following are Oracle9i's Java components:

- **JDBC Driver**

Oracle provides JDBC as the core programmatic interface for accessing Oracle9i databases. There are three JDBC drivers provided by Oracle: two for client access and one for server access.

- The JDBC OCI Driver is used by Java applications.
- The JDBC Thin driver is primarily used by Java applets.
- The Oracle JDBC Server-side Thin driver offers the same functionality as the client-side JDBC Thin driver and is used primarily by Java classes running on the Java VM of the database server to access a remote database.
- The JDBC Server-side Internal driver is a server-side driver that is used by Java classes running on the Java VM of the database server.

- SQLJ translator

SQLJ acts like a preprocessor that translates embedded SQL in the SQLJ program file into a Java source file with JDBC calls. It gives programmers a higher level of programmatic interface for accessing databases.

- Java runtime environment

A Java VM based on that of the JDK is integrated into the database server that enables the running of Java classes. It comes with a set of supporting services such as the library manager, which manages Java classes stored in the database.

- Java Servlet support

The Java Servlet API 2.2 has been implemented to support the deployment of Java servlets in the Oracle JVM. The Servlet API provides important internationalization support to build a multilingual Java servlet. A JavaServer Page (JSP) compiler is also provided to compile JSPs into Java servlets.

- CORBA support

In addition to the Java runtime environment, Oracle integrates the CORBA Object Request Broker (ORB) into the database server, and makes the database a CORBA server. Any CORBA client can call the Java CORBA objects published to the ORB of the database server.

- EJB support

The Enterprise Java Bean version 1.1 container is built into the database server to provide a platform to develop and deploy EJBs.

JDBC

This section describes the following:

- [Accessing SQL CHAR Datatypes Using JDBC](#)
- [Accessing SQL NCHAR Datatypes Using JDBC](#)
- [Using the oracle.sql.CHAR Class](#)
- [NLS Restrictions](#)

Oracle JDBC drivers provide globalization support by allowing you to retrieve data from or insert data into columns of the SQL `CHAR` datatypes and the SQL `NCHAR` datatypes of an Oracle*i* database. Because Java strings are UTF-16 encoded (16-bit Unicode) for JDBC programs, the target character set on the client is always UTF-16. For data stored in the `CHAR`, `VARCHAR2`, `LONG`, and `CLOB` datatypes, JDBC

transparently converts the data from the database character set to UTF-16. For Unicode data stored in the NCHAR, NVARCHAR2, and NCLOB datatypes, JDBC transparently converts the data from the national character set to UTF-16.

Following are a few examples of commonly used Java methods for JDBC that rely heavily on NLS character set conversion:

- `java.sql.ResultSet`'s method `getString()` returns values from the database as Java strings.
- `java.sql.ResultSet`'s method `getUnicodeStream()` returns values as a stream of Unicode characters.
- `oracle.sql.CLOB`'s method `getSubString()` returns the contents of a CLOB as a Unicode stream.
- `oracle.sql.CHAR`'s methods `getString()`, `toString()`, and `getStringWithReplacement()`.

At database connection time, the JDBC Class Library sets the server NLS_ LANGUAGE and NLS_TERRITORY parameters to correspond to the locale of the Java VM that runs the JDBC driver. This operation is performed on the JDBC OCI and JDBC Thin drivers only, and ensures that the server and the Java client communicate in the same language. As a result, Oracle error messages returned from the server are in the same language as the client locale.

Accessing SQL CHAR Datatypes Using JDBC

To insert a Java string to a database column of a SQL CHAR datatype, you may use the `PreparedStatement.setString()` method to specify the bind variable, and Oracle's JDBC drivers transparently convert the Java string to the database character set. An example for binding a Java string `ename` to a `VARCHAR2` column `ename` is shown below.

```
int empno = 12345;
String ename = "\uFF2A\uFF4F\uFF45";
PreparedStatement pstmt =
    conn.prepareStatement ("INSERT INTO emp (empno, ename) VALUES(?,?);
pstmt.setInt(1, empno);
pstmt.setString(2, ename);
pstmt.execute();
pstmt.close();
```

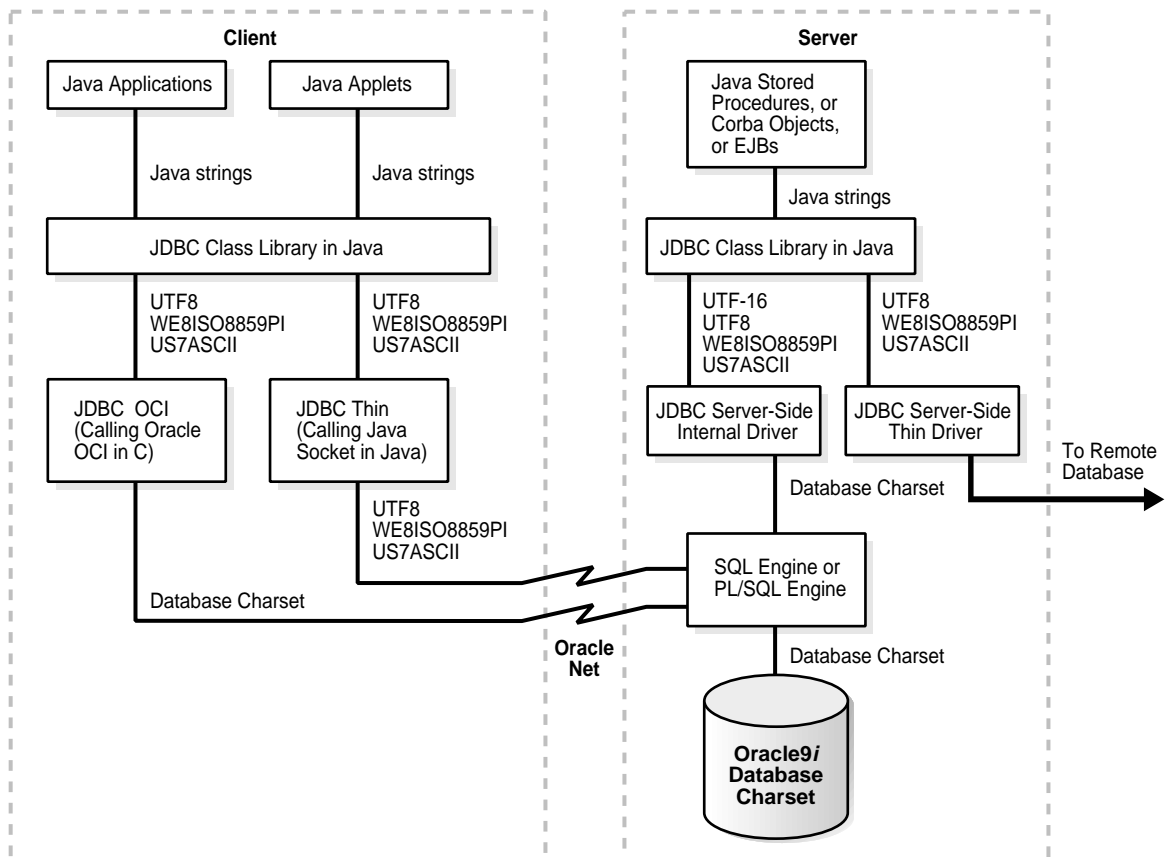
For data stored in the SQL CHAR datatypes, the techniques that Oracle's drivers use to perform character set conversion for Java applications depend on the character

set that the database uses. The simplest case is where the database uses a US7ASCII or WE8ISO8859P1 character set. In this case, the driver converts the data directly from the database character set to UTF-16, which is used in Java applications.

If you are working with databases that employ a non-US7ASCII or non-WE8ISO8859P1 character set (for example, Japanese or Korean), then the driver converts the data, first to UTF8, then to UTF-16. The following sections detail the conversion paths for different JDBC drivers.

Figure 9-1 presents a graphical view of how data is converted in JDBC drivers.

Figure 9-1 JDBC Data Conversion



JDBC Class Library

The JDBC Class Library is a Java layer that implements the JDBC interface. Java applications, applets, and stored procedures interact with this layer. The library always accepts US7ASCII, UTF8 or WE8ISO8859P1 encoded string data from the input stream of the JDBC drivers. It also accepts UTF-16 for the JDBC server-side driver. The JDBC Class Library converts the input stream to UTF-16 before passing it to the client applications. If the input stream is in UTF8, the JDBC Class Library converts the UTF8 encoded string to UTF-16 by using the bit-wise operation defined in the UTF8-to-UTF-16 conversion algorithm. If the input stream is in US7ASCII or WE8ISO8859P1, it converts the input string to UTF-16 by casting the bytes to Java characters. This is based on the first 128 and 256 characters of UTF-16 corresponding to the US7ASCII and WE8ISO8859P1 character sets, respectively. Treating WE8ISO8859P1 and US7ASCII separately improves the performance for commonly used single-byte clients by eliminating the bit-wise conversion to UTF8.

JDBC OCI Driver

In the case of a JDBC OCI driver installation, there is a client-side character set as well as a database character set. The client character set is determined at client installation by the value of the `NLS_LANG` environment variable. The database character set is determined at database creation. The character set used by the client can be different from the character set used by the database on the server. So, when performing character set conversion, the JDBC OCI driver has to take three factors into consideration:

- The database character set and language
- The client character set and language
- The Java application's character set

The JDBC OCI driver transfers the data from the server to the client in the character set of the database. Depending on the value of the `NLS_LANG` environment variable, the driver handles character set conversions in one of two ways:

- If the value of `NLS_LANG` is not specified, or if it is set to the US7ASCII or WE8ISO8859P1 character set, then the JDBC OCI driver uses Java to convert the character set from US7ASCII or WE8ISO8859P1 directly to UTF-16 in the JDBC class library.
- If the value of `NLS_LANG` is set to a non-US7ASCII or non-WE8ISO8859P1 character set, then the driver changes the value of the `NLS_LANG` parameter on the client to UTF8. This happens automatically and does not require any user-intervention. OCI uses the value of `NLS_LANG` to convert the data from the

database character set to UTF8; the OCI JDBC driver then passes the data to the JDBC class library where the UTF8 data is converted to UTF-16.

JDBC Thin Driver

If your applications or applets use the JDBC Thin driver, then there is no Oracle client installation. Because of this, the OCI client conversion routines in C are not available. In this case, the client conversion routines are different from the JDBC OCI driver.

If the database character set is US7ASCII or WE8ISO8859P1, the data is transferred to the client without any conversion. The driver then converts the character set to UTF-16 in Java.

If the database character set is something other than US7ASCII or WE8ISO8859P1, then the server first translates the data to UTF8 before transferring it to the client. On the client, the JDBC Thin driver converts the data to UTF-16 in Java.

JDBC Server-side Internal Driver

For Java classes running in the Java VM of the Oracle9i Server, the JDBC Server-side Internal driver is used to talk to the SQL engine or the PL/SQL engine for SQL processing. Because the JDBC Server-side Internal driver is running in the same address space as the Oracle server process, it makes a local function call to the SQL engine or the PL/SQL engine. Data sent to or returned from the SQL engine or the PL/SQL engine will be encoded in the database character set. If the database character set is US7ASCII, WE8ISO8859P1, or UTF8, no conversion is performed in the JDBC Server-side Internal driver, and the data is passed to or from the JDBC Class Library as is. Otherwise, the JDBC Server-side Internal driver converts the data from the database character set to UTF-16 before passing it to and from the class library. The class library does not need to do any conversion in this case.

Accessing SQL NCHAR Datatypes Using JDBC

JDBC allows Java programs to access columns of the SQL NCHAR datatypes in an Oracle9i database. The data conversion path for the SQL NCHAR datatypes is different from that of the SQL CHAR datatypes. All Oracle JDBC drivers convert data in the SQL NCHAR column from the national character set, which is either UTF8 or AL16UTF16, directly to UTF-16 encoded Java strings. In a Java program, you may bind a Java string `ename` to a `NVARCHAR2` column `ename` as follows:

```
int empno = 12345;
String ename = "\uFF2A\uFF4F\uFF45";
oracle.jdbc.OraclePreparedStatement pstmt =
```

```
(oracle.jdbc.OraclePreparedStatement)
conn.prepareStatement("INSERT INTO emp (empno, ename) VALUES (?, ?)");
pstmt.setFormOfUse(2, oracle.jdbc.OraclePreparedStatement.FORM_NCHAR);
pstmt.setInt(1, empno);
pstmt.setString(2, ename);
pstmt.execute();
pstmt.close();
```

See Also: [Chapter 6, "Unicode Programming"](#) for more information about programming against the SQL NCHAR datatypes

Using the oracle.sql.CHAR Class

The `oracle.sql.CHAR` class has special functionality for NLS conversion of character data. A key attribute of the `oracle.sql.CHAR` class, and a parameter always passed in when an `oracle.sql.CHAR` object is constructed, is the NLS character set used in presenting the character data. Without a known character set, the bytes of data in the `oracle.sql.CHAR` object are meaningless.

Retrieving Data to oracle.sql.CHAR Class

When you call the `OracleResultSet.getCHAR()` method to get a bind variable as an `oracle.sql.CHAR` object, JDBC constructs and populates the `oracle.sql.CHAR` objects after character data has been read from the database.

The `oracle.sql.CHAR` class provides the following methods for converting character data to strings:

- `getString()`
Converts the sequence of characters represented by the `oracle.sql.CHAR` object to a string, returning a Java `String` object. If the character set is not recognized (that is, if you entered an invalid `OracleID`), then `getString()` throws a `SQLException`.
- `toString()`
Identical to `getString()`, but if the character set is not recognized (that is, if you entered an invalid `OracleID`), then `toString()` returns a hexadecimal representation of the `oracle.sql.CHAR` data and does *not* throw a `SQLException`.
- `getStringWithReplacement()`

Identical to `getString()`, except a default replacement character replaces characters that have no Unicode representation in the character set of this `oracle.sql.CHAR` object. This default character varies from character set to character set, but is often a question mark.

The `oracle.sql.CHAR` in Oracle Object Types

In Oracle9i, JDBC drivers support Oracle object types. Oracle objects are always sent from database to client as an object represented in the database character set. That means the data conversion path in [Figure 9-1, "JDBC Data Conversion"](#), does not apply to Oracle object access. Instead, the `oracle.sql.CHAR` class is used for passing string data from the database to the client. The following is an example of an object type created using SQL:

```
CREATE TYPE PERSON_TYPE AS OBJECT (NAME VARCHAR2(30), AGE NUMBER);
CREATE TABLE EMPLOYEES (ID NUMBER, PERSON PERSON_TYPE);
```

The Java class corresponding to this object type can be constructed as follows:

```
public class person implement SqlData
{
    oracle.sql.CHAR name;
    oracle.sql.NUMBER age;
    // SqlData interfaces
    getSqlType() {...}
    writeSql(SqlOutput stream) {...}
    readSql(SqlInput stream, String sqltype) {...}
}
```

The `oracle.sql.CHAR` class is used here to map to the `NAME` attributes of the Oracle object type, which is of `VARCHAR` type. JDBC populates this class with the byte representation of the `VARCHAR` data in the database and the character set object corresponding to the database character set. The following code retrieves a person object from the `people` table,

```
TypeMap map = ((OracleConnection)conn).getTypeMap();
map.put("PERSON_TYPE", Class.forName("person"));
conn.setTypeMap(map);

.      .      .
.      .      .

ResultSet rs = stmt.executeQuery("SELECT PERSON FROM EMPLOYEES");
rs.next();
person p = (person) rs.getObject(1);
oracle.sql.CHAR sql_name = p.name;
String java_name = sql_name.getString();
```

The `getString()` method of the `oracle.sql.CHAR` class converts the byte array from the database character set to UTF-16 by calling Oracle's Java data conversion classes and return a Java string. For the `rs.getObject(1)` call to work, the `SqlData` interface has to be implemented in the class `person`, and the `Typemap` map has to be set up to indicate the mapping of the object type `PERSON_TYPE` to the Java class.

NLS Restrictions

CHAR and VARCHAR2 Data Size Restriction With the Thin Driver

If the database character set is neither ASCII (US7ASCII) nor ISO-LATIN-1 (WE8ISO8859P1), then the Thin driver must impose size restrictions for `CHAR` and `VARCHAR2` bind parameters that are more restrictive than normal database size limitations. This is necessary to allow for data expansion during conversion.

The Thin driver checks `CHAR` or `VARCHAR2` bind sizes when the `setXXX()` method is called. If the data size exceeds the size restriction, then the driver throws a SQL exception (ORA-17070 "Data size bigger than max size for this type") from the `setXXX()` call. This limitation is necessary to avoid the chance of data corruption whenever an NLS conversion occurs and increases the length of the data. This limitation is enforced when you are doing all the following:

- Using the Thin driver
- Using binds (not defines)
- Using `CHAR` or `VARCHAR2` datatypes
- Connecting to a database whose character set is neither ASCII (US7ASCII) nor ISO-Latin-1 (WE8ISO8859P1)

Role of NLS Ratio When the database character set is neither US7ASCII nor WE8ISO8859P1, the Thin driver converts Java UTF-16 characters to UTF-8 encoding bytes for `CHAR` or `VARCHAR2` binds. The UTF-8 encoding bytes are then transferred to the database, and the database converts the UTF-8 encoding bytes to the database character set encoding.

This conversion to the character set encoding might result in a size increase. The NLS ratio for a database character set indicates the maximum possible expansion in converting from UTF-8 to the character set:

NLS ratio = maximum character size in the database character set

Size Restriction Formulas Table 9–1 shows the database size limitations for CHAR and VARCHAR2 data, and the Thin driver size restriction formulas for CHAR and VARCHAR2 binds. Database limits are in bytes. Formulas determine the maximum size of the UTF-8 encoding, in bytes.

Table 9–1 Maximum CHAR and VARCHAR2 Bind Sizes, Thin Driver

Oracle Version	Datatype	Maximum Size Allowed by Database (In Bytes)	Formula for Thin Driver Maximum Bind Size (In UTF8 Bytes)
Oracle8, Oracle8i, and Oracle9i	CHAR	2000	$2000 / \text{NLS_ratio}$
Oracle8, Oracle8i, and Oracle9i	VARCHAR2	4000	$4000 / \text{NLS_ratio}$

The formulas guarantee that after the data is converted from UTF-8 to the database character set, the size will not exceed the database maximum size.

The number of UTF-16 characters that can be supported is determined by the number of bytes per character in the data. All ASCII characters are one byte long in UTF-8 encoding. Other character types can be two or three bytes long.

NLS Ratios and Calculated Size Restrictions for Common Character Sets Table 9–2 lists the NLS ratios of some common server character sets, then shows the Thin driver maximum bind sizes for CHAR and VARCHAR2 data for each character set, as determined by using the NLS ratio in the appropriate formula.

Maximum bind sizes are for UTF-8 encoding, in bytes.

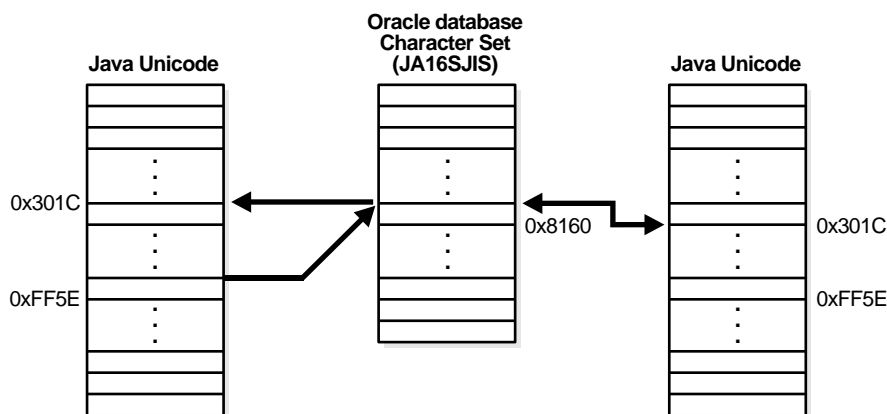
Table 9–2 NLS Ratio and Size Limits for Common Server Character Sets

Server Character Set	NLS Ratio	Thin Driver Maximum VARCHAR2 Bind Size (UTF-8 Bytes)	Thin Driver Maximum CHAR Bind Size (UTF-8 Bytes)
WE8DEC	1	4000	2000
JA16SJIS	2	2000	2000
JA16EUC	3	1333	1333

Character Integrity Issues in an NLS Environment

Oracle JDBC drivers perform character set conversions as appropriate when character data is inserted into or retrieved from the database. In other words, the drivers convert Unicode characters used by Java clients to Oracle database character set characters, and vice versa. Character data making a round trip from the Java Unicode character set to the database character set and back to Java can suffer some loss of information. This happens when multiple Unicode characters are mapped to a single character in the database character set. An example would be the Unicode full-width tilde character (0xFF5E) and its mapping to Oracle's JA16SJIS character set. The round trip conversion for this Unicode character results in the Unicode character 0x301C, which is a wave dash (a character commonly used in Japan to indicate range), not a tilde.

Figure 9–2 Character Integrity



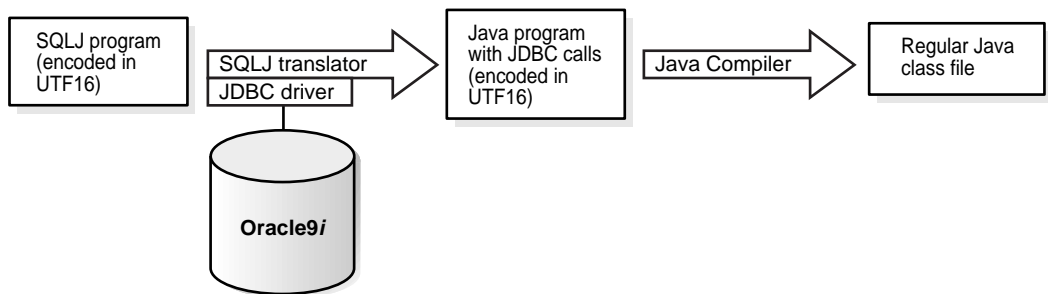
This issue is not a bug in Oracle's JDBC. It is an unfortunate side effect of the ambiguity in character mapping specification on different operating systems. Fortunately, this problem affects only a small number of characters in a small number of Oracle character sets such as JA16SJIS, JA16EUC, ZHT16BIG5, and KO16KS5601. The workaround is to avoid making a full round-trip with these characters.

SQLJ

SQLJ is a SQL-to-Java translator that translates embedded SQL statements in a Java program into the corresponding JDBC calls regardless of which JDBC driver is used. It also provides a callable interface that the Oracle9i database server uses to transparently translate the embedded SQL in server-side Java programs. SQLJ by itself is a Java application that reads the SQLJ programs (Java programs containing embedded SQL statements) and generates the corresponding Java program files with JDBC calls. There is an option to specify a checker to check the embedded SQL statements against the database at translation time. The `javac` compiler is then used to compile the generated Java program files to regular Java class files.

Figure 9–3 presents a graphical view of how the SQLJ translator works.

Figure 9–3 Using the SQLJ Translator



Using Unicode characters in SQLJ programs

SQLJ enables multilingual Java application development by allowing SQLJ files encoded in different encoding schemes (those supported by the JDK). In the diagram above, a UTF-16 encoded SQLJ program is being passed to the SQLJ translator and the Java program output is also encoded in UTF-16. SQLJ preserves the encoding of the source in the target. To specify the encoding of the source, use the `-encoding` option as follows:

```
sqlj -encoding Unicode source_file
```

Unicode notation `\uXXXX` (which is referred to as a Unicode escape sequence) can be used in embedded SQL statements for characters that cannot be represented in the encoding of the SQLJ program file. This enables you to specify multilingual object names in the SQL statement without using a UTF-16 encoded SQLJ file. The

following SQLJ code shows the usage of Unicode escape sequences in embedded SQL as well as in a string literal.

```
int empno = 12345;
String name ename = "\uFF2A\uFF4F\uFF45";
double raise = 0.1;

#sql {INSERT INTO E\u0063\u0064 (ename, empno) VALUES (:ename, :empno)};
#sql { update EMP set SAL = :(getNewSal(raise, ename))
WHERE ename = :ename;
```

See Also: ["Multilingual Demo Applications in SQLJ"](#) on page 9-31
for an example of SQLJ usage for a multilingual Java application

Using the oracle.sql.NString class

In Oracle9i, the `oracle.sql.NString` class is introduced in SQLJ to support the NVARCHAR2, NCHAR, and NCLOB Unicode datatypes. You may declare a bind an NCHAR column using a Java object of the `oracle.sql.NString` type, and use it in the embedded SQL statements in your SQLJ programs.

```
int empno = 12345;
oracle.sql.NString ename = new oracle.sql.NString ("\uFF2A\uFF4F\uFF45");
double raise = 0.1;
#
#sql {INSERT INTO E\u0063\u0064 (ENAME, EMPNO) VALUES (:ename, :empno)};
sql { UPDATE emp SET sal = :(getNewSal(raise, ename))
WHERE ename = :ename;
```

This example binds the `ename` object of the `oracle.sql.NString` datatype to the `ename` database NVARCHAR2 column.

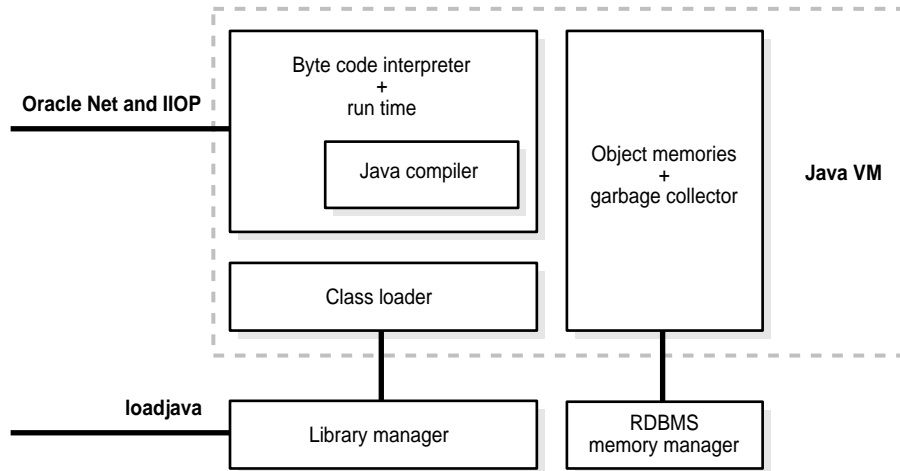
See Also: [Chapter 6, "Unicode Programming"](#) for more details on the SQL NCHAR datatypes support in SQLJ

Java Virtual Machine

The Oracle9i Java VM base is integrated into the database server to enable the running of Java classes stored in the database. Oracle9i allows you to store Java class files, Java or SQLJ source files and Java resource files into the database, to publish the Java entry points to SQL so that it can be called from SQL or PL/SQL, and to run the Java byte code.

In addition to the engine that interprets Java byte code, the Oracle Java VM includes the core run-time classes of the JDK. The components of the Java VM are depicted in Figure 9–4.

Figure 9–4 Components of Oracle's Java Virtual Machine



The Java VM provides an embedded Java class loader that locates, loads, and initializes locally stored Java classes in the database, and a byte code compiler which translates standard Java programs into standard Java `.class` binary representation. A library manager is also included to manage Java program, class, and resource files as schema objects known as **library units**. It not only loads and manages these Java files in the database, but also maps Java name space to library units. For example:

```

public class Greeting
{
    public String Hello(String name)
    {
        return ("Hello" + name + "!");
    }
}
  
```

After the preceding Java code is compiled, it is loaded into the database as follows:

```
loadjava Greeting.class
```

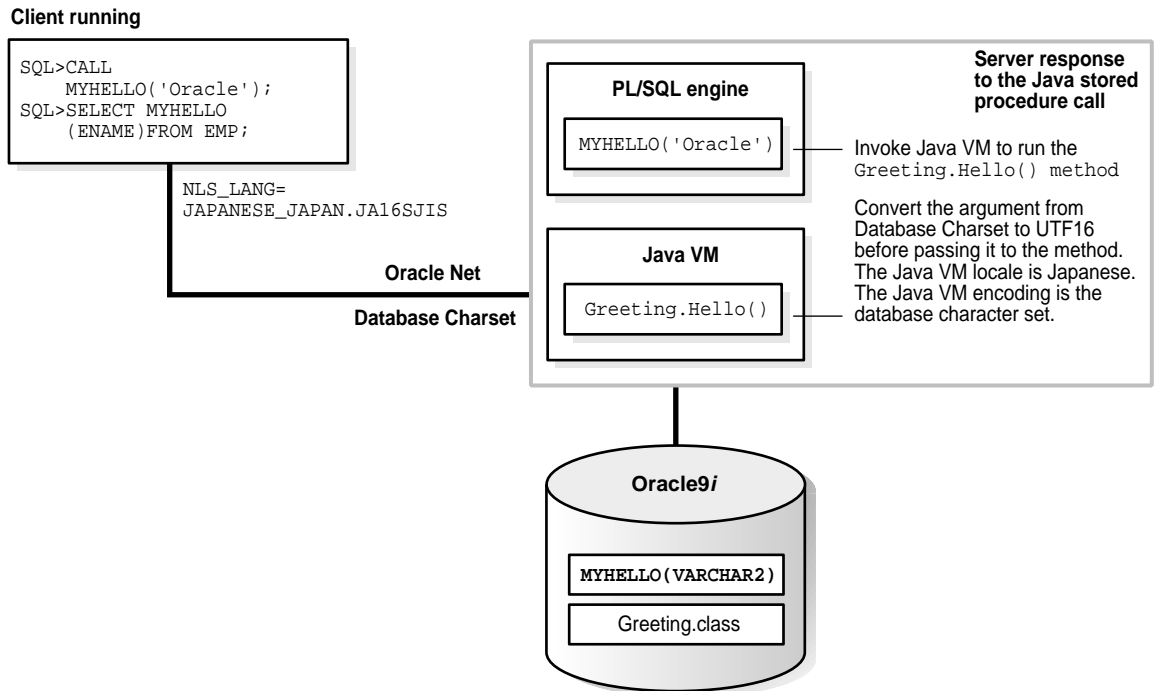
As a result, a library unit called `Greeting`, is created as a schema object in the database. Class and method names containing characters that cannot be represented in the database character set are handled by generating a US7ASCII library unit name and mapping it to the real class name stored in a `RAW` column. This allows the class loader to find the library unit corresponding to the real class name when Java programs run in the server. In other words, the library manager and the class loader support class names or method names outside the namespace of the database character set.

Java Stored Procedures

A Java stored procedure or function requires that the library unit of the Java classes implementing it already be present in the database. Using the `Greeting` library unit example in the previous section, the following call specification DDL publishes the method `Greeting.Hello()` as a Java stored function:

```
CREATE FUNCTION MYHELLO(NAME VARCHAR2) RETURN VARCHAR2
  AS LANGUAGE JAVA NAME
  'Greeting.Hello(java.lang.String) return java.lang.String';
```

The DDL maps the Java methods, parameter types and return types to the SQL counterparts. To the users, the Java stored function has the same calling syntax as any other PL/SQL stored functions. Users can call the Java stored procedures the same way they call any PL/SQL stored procedures. [Figure 9-5](#) depicts the runtime environment of a stored function.

Figure 9–5 Running of Java Stored Procedures

The Java entry point, `Greeting.Hello()`, is called by invoking the proxy PL/SQL `MYHELLO()` from the client. The server process serving the client runs as a normal PL/SQL stored function and uses the same syntax. The PL/SQL engine takes a call specification for a Java method and calls the Java VM. Next, it passes the method name of the Java stored function and the argument to the Java VM for execution. The Java VM takes control, calls the SQL to Java using code to convert the `VARCHAR2` argument from the database character set to UTF-16, loads the class `Greeting`, and runs the method `Hello()` with the converted argument. The string returned by `Hello()` is then converted back to the database character set and returned as a `VARCHAR2` string to the caller.

The globalization support that enables deployment and development of internationalized Java stored procedures includes:

1. The strings in the arguments of Java stored procedures are automatically converted from SQL data types (in the database character set) to UTF-16-encoded Java strings.
2. The default Java locale of the Java VM follows the language setting (defined by the `NLS_LANGUAGE` and `NLS_TERRITORY` database parameters) of the current database session propagated from the `NLS_LANG` environment variable of the client. A mapping of Oracle language and territory names to Java locale names is in place for this purpose. In additions, the default encoding of the Java VM follows the database character set.
3. The `loadjava` utility supports loading of Java and SQLJ source files encoded in any encoding supported by the JDK. The content of the Java or SQLJ program is not limited by the database character set. Unicode escape sequences are also supported in the program files.

Note: The entry method name and class name of a Java stored procedure has to be in the database character set because it has to be published to SQL as DDL.

Java Servlets and Java Server Pages

You can write a Java servlet or a JavaServer Page (JSP) and deploy it on an Oracle9i database. Oracle9i provides a servlet engine and JSP compiler for the deployment of Java servlets and JSPs.

- The Oracle Servlet Engine (OSE) that implements the Servlet 2.2 API which provides the internationalization support for writing a multilingual Java servlet.
- The Oracle JSP compiler (OJSP) supports the JSP 1.1 and provides internationalization support for writing a multilingual JSP.

When a HTTP requests come to invoke a Java servlet in OSE, the Oracle HTTP Server directs the request to the database, the database invokes OSE in the context of a database session, OSE locates the requested Java servlet and dispatches to it with the `HTTPServletRequest` object and the `HTTPServletResponse` object.

To deploy a Java servlet or JSP to the database, follow the steps below:

1. Write the Java servlets and JSPs.
2. Compile the JSPs into Java servlets.
3. Load the Java servlets into the database using the `loadjava` utility

4. Publish the Java servlets using the `httpublish` utility. You publish a Java servlet with a virtual path for the servlet.

See Also: *Oracle9i Servlet Engine Developer's Guide* for information about managing Java servlets in OSE

A Java servlet or JSP receives HTTP requests from a browser, processes the request, and generates an HTTP response back to the browser.

The following sections describe the things you should consider when programming a Java servlet or JSP to support multiple languages.

Determining the Desired Locale of a User

To present the user interface in the user's desired language, applications need to detect his or her desired locale and construct HTML content in the desired language and use the correct cultural conventions. Both JSPs and Java servlets can use the `HttpServletRequest.getLocale()` method of the Servlet API to get the Java locale corresponding to the Accept-Language HTTP header and use it as the desired locale. Once the desired locale is found, set the Java locale as the default Java locale to direct all locale-sensitive Java objects functions to behave accordingly.

```
Locale.setDefault(userLocale);
```

The default Java locale is used for all Java threads. To ensure that different locales are used on different threads, specify the desired locale for each Java object.

Tagging the HTML Output with an Encoding

The encoding of an HTML page is a very important piece of information to the browser and your applications. The browser needs to know so that it can use correct font and mapping tables for displaying pages, and applications need to know so they can safely assume the encoding of form input data and query strings.

You can tag the HTTP header by calling the `setContentType()` method of the Servlet API. The following `doGet()` function shows how this method should be called to specify UTF-8 as the encoding of the HTML output.

```
public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException
{
    // generate the MIME type and character set header
    res.setContentType("text/html; charset=utf-8");
}
```

```
// generate the HTML page
Printwriter out = res.getWriter();
out.println("<HTML>");
.. .. .
out.println("</HTML>");
}
```

Note that the `setContentType()` method should be called before the `getWriter()` method because the `getWriter()` method initializes an output stream writer using the character set specified in the `setContentType()` method.

For JSPs, you can tag the encoding of an HTML page using the `contentType` page directive. An example is shown below.

```
<%@ page contentType="text/html; charset=utf-8" %>
```

The character set of the `contentType` page directive describes the encoding of the JSP page as well as the encoding of the HTML page sent to the browser.

Decoding Form Input and Encoding Query String

In most JSP and servlets engines including OSE, the Servlet API implementation assumes that incoming form input is in ISO-8859-1 encoding. As a result, when the `HttpServletRequest.getParameter()` API is called, all data of the input text is decoded and the decoded input is converted from ISO-8859-1 to UTF-16 and returned as a Java string. The Java string returned is incorrect if the encoding of the HTML form is not in ISO-8859-1. However, you can solve this. When the JSP or Java servlet receives form input or query strings, it needs to convert them back to the original form, and then convert the original form to a Java string based on the correct encoding.

```
String orig = request.getParameter("name");
String real = new String(orig.getBytes("ISO8859_1"), "UTF8");
```

In the above example, the Java string `real` will be initialized to store correct characters from a UTF-8 form input.

If a query string is constructed in a JSP or Java servlet, all 8-bit bytes must be encoded using their hexadecimal values prefixed by a percent sign as described above. The following code shows you how to encode a Java string into its hexadecimal representation in UTF-8.

```
byte[] htmlBytes = queryString.getBytes("UTF8");
for (int i= 0; i < htmlBytes.length; i++)
{
```

```
if ((htmlBytes[i] & 0xff) > 0x7f)
    queryString += "%" + Long.toHexString
        ((long)(htmlBytes[i] & 0xff));
else
    queryString += new String(htmlBytes,i, 1,"ISO8859_1");
}
```

CORBA and EJB

Visigenic's CORBA Object Request Broker (ORB) is integrated into the database server to make it a Java CORBA object and EJB server running the IIOP protocol. CORBA support also includes a set of supporting services that enables the deployment of CORBA objects to the database.

See Also: *Oracle9i CORBA Developer's Guide and Reference.*

CORBA ORB

The CORBA ORB is written in Java and includes an IIOP interpreter and the object adapter. The IIOP interpreter processes the IIOP message by invoking the object adapter to look for the CORBA object being activated and load it into the memory, and running the object method specified in the message.

A couple of CORBA objects are predefined. The `LoginServer` object is used for explicit session log in, and the `PublishContext` object is used to resolve a published CORBA object name to the corresponding `PublishedObject`.

CORBA objects implemented in Java in Oracle9i are required to be loaded and then published before the client can reference it. `Publish` is a Java written utility that publishes a CORBA object to the ORB by creating an instance of `PublishedObject` which represents and activates the CORBA object, and binding the input (`CosNaming`) name to the published object.

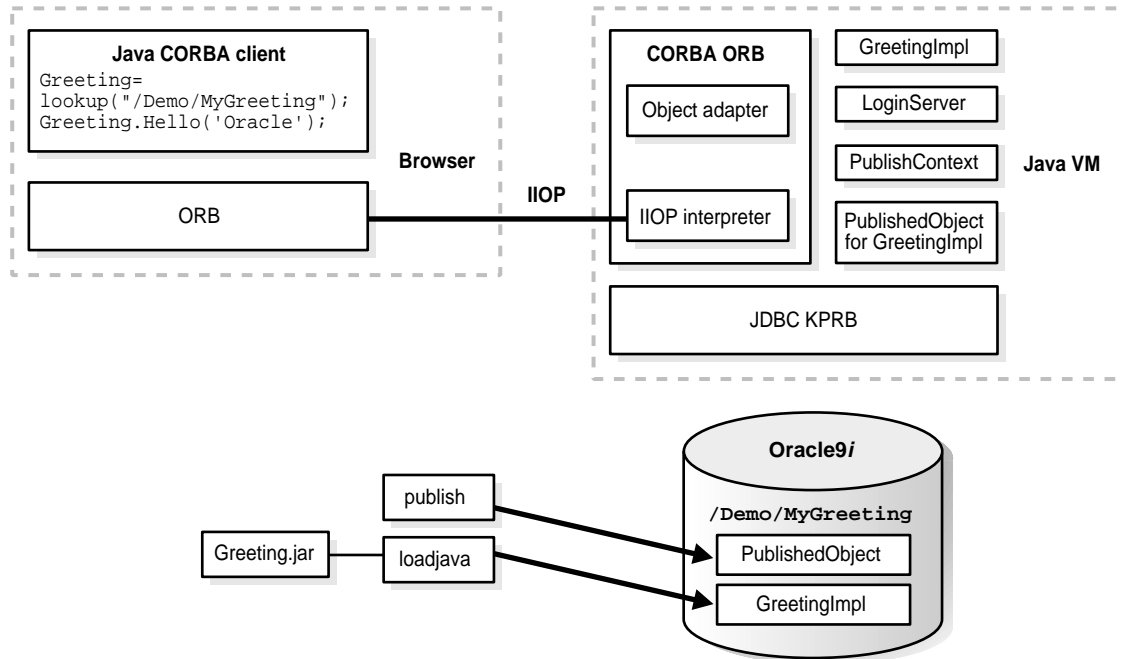
Oracle9i implements the `CosNaming` standard for specifying CORBA object names. `CosNaming` provides a directory-like structure that is a context for binding names to CORBA objects. A new JNDI URL, `sess_iiop:`, is created, and indicates a session-based IIOP connection for a CORBA object. A name for a CORBA object in the local database can be published as:

```
sess_iiop://local:2222:ORCL/Demo/MyGreeting
```

where 2222 is the port number for receiving IIOP requests, ORCL is the database instance identifier and `/Demo/MyGreeting` is the name of the published object. The namespace for CORBA objects in Oracle9i is limited to US7ASCII characters.

Figure 9–6 presents a graphical view of the components in a CORBA environment:

Figure 9–6 Components Supporting CORBA



Java CORBA Object

The CORBA objects for Oracle9i can only be written in Java and they run on the Java VM of the database. The CORBA client can be written in any language the standard supports. An interface definition language (IDL) file that identifies the CORBA objects and their interfaces will be compiled with the `idl2java` translator to generate the stub for the client and the skeleton code for the CORBA server objects. CORBA object programmers are required to program the implementation classes of the CORBA objects defined in the IDL in Java by extending the skeleton classes generated and load them to the database together with the skeleton code.

Greeting.idl

```
Module Demo
{
    interface Greeting
```

```

    {
        wstring Hello(string str);
    };
};

```

```
>idl2java Greeting.IDL
```

```
Creating:
```

```

    Demo/Greeting.java
    Demo/GreetingHolder.java
    Demo/GreetingHelper.java
    Demo/_GreetingImpBase.java

```

GreetingImpl.java

```

public class GreetingImpl
extends _GreetingImpBase
implements ActivatableObject
{
    public GreetingImpl (String name)
    {
        super(name);
    }
    public GreetingImpl()
    {
        super();
    }
    public org.omg.CORBA.Object
    _intializeAuroraObject()
    {
        return this
    }
    public String Hello(String str)
    {
        return "Hello" + str;
    }
}

```

In the above code, the CORBA object `Greeting` has been implemented with a method called `Hello()`. The CORBA standard defines the `wstring` data type to pass multibyte strings via CORBA/IIOP, and the Visigenic ORB implements the `wstring` data type as a Unicode string. If the `string` datatype is specified instead, the parameter passed into the `Hello()` method is assumed to be a single byte. The `wstring` data type enables the development of multilingual CORBA objects. The

implementation class for `Greeting` extends the skeleton class `_GreetingImplBase` generated by `idl2java`.

Once the CORBA object has been implemented, the below example shows the steps involved in loading the Java object implementation classes into the database and publishing the Java CORBA object using the `CosNaming` convention.

```
loadjava -user scott/tiger -grant public Greeting.jar
publish -user scott -password tiger -service
        sess_iiop://local:2222:orcl/Demo/MyGreeting
        Demo.GreetingImpl Demo.GreetingHelper
```

Assume that all Java classes (implementation and helper classes) required to implement the `Greeting` object are in the `Greeting.jar` file. They are loaded to the database as `public`, and the implementation class is published to the database. The name of the published object is `/Demo/MyGreeting`, and it is used in the client code to reference this CORBA object.

Java CORBA Client

Clients accessing a CORBA object in the database require an ORB and authentication from the database where the object is stored. The following is an excerpt of a client code in Java accessing the `Greeting` object. The ORB is initialized when the CORBA object is first activated by means of Oracle's implementation of the Java Native Directory Interface (JNDI).

```
import java.util.Hashtable;
import javax.naming.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class Client
{
    public static void main(String args[]) throws Exception
    {
        Hashtable environment = new Hashtable();
        environment.put(javax.naming.Context.URL_PKG_PREFIXES,
            "oracle.aurora.jndi");
        environment.put(Context.SECURITY_PRINCIPAL, "scott");
        environment.put(Context.SECURITY_CREDENTIALS, "tiger");
        environment.put(Context.SECURITY_AUTHENTICATION,
            ServiceCtx.NON_SSL_CREDENTIALIAL);

        Context ic = new InitialContext(environment);
        Greeting greet = (Greeting)
            ic.lookup("sess_iiop://local:2222:ORCL/Demo/MyGreeting");
```

```
        System.out.println(greet.Hello(arg[0]));  
    }  
}
```

The database is a secure environment, so Java clients must be authenticated before they can access CORBA objects, and the locale of the Java VM running the CORBA object is initialized when the session running the object is authenticated. To access a CORBA object, you can use explicit or implicit authentication:

- **Implicit authentication**

The client can initialize the service context object with its user name and password as shown in the above code. The default locale of the client Java VM is implicitly stored in the service context object and passed to the server ORB in the first IIOP request. The server Java VM locale is initialized with the same locale as the client.

- **Explicit authentication**

The client can call the `authenticate()` method of the `Login` object to access the `LoginServer` CORBA object in the server. The `LoginServer` object can be accessed without being authenticated. The `authenticate()` method accepts user name, password, role and Java locale as arguments. If the Java locale argument is not provided, the default locale of the Java VM in the server will be initialized to the database language defined by the `NLS_LANGUAGE` and `NLS_TERRITORY` database parameters.

Enterprise Java Beans

In addition to CORBA objects, Oracle provides tools and an environment for developing and deploying EJBs in the Oracle9i server. An EJB is called using the IIOP protocol provided for CORBA support, and hence shares a lot of similarities with the CORBA object. An EJB is defined in the EJB descriptor, which specifies the home interface, remote interface, home name and allowed identities of the EJB among other things.

See Also: *Oracle9i Enterprise JavaBeans Developer's Guide and Reference*

The following shows the EJB descriptor for `GreetingBean`, which is functionally equivalent to the CORBA object `Greeting` described earlier.

```
SessionBean GreetingServer.GreetingBean  
{
```

```
BeanHomeName = "Demo/MyGreeting";
RemoteInterfaceClassName = hello.Greeting;
HomeInterfaceClassName = hello.GreetingHome;
AllowedIdentities = { PUBLIC };
RunAsMode = CLIENT_IDENTITY;
TransactionAttribute = TX_SUPPORTS;
}
```

An EJB descriptor can be in any encoding supported by the JDK. However, only the `AllowedIdentities` field can be non-US7ASCII. There are two ways you can specify non-US7ASCII `AllowedIdentities`:

- Use the encoding of the non-US7ASCII character set for the EJB descriptor file and specify the `-encoding` command line argument to tell `ejbdeploy` the encoding of the input file.
- Use the corresponding Unicode escape sequence to represent the non-US7ASCII identities.

The implementation class for the EJB is in `GreetingBean.java` package `GreetingServer`;

```
import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

public class GreetingBean implements SessionBean
{
    // Methods of the Greeting interface
    public String Hello (String str) throws RemoteException
    {
        return "Hello" + str;
    }
    // Methods of the SessionBean
    public void ejbCreate () throws RemoteException, CreateException {}
    public void ejbRemove() {}
    public void setSessionContext (SessionContext ctx) {}
    public void ejbActivate () {}
    public void ejbPassivate () {}
}
```

Note that all strings passed to the EJB as arguments and returned from the EJB as function values are UTF-16 encoded Java strings.

An EJB resembles a CORBA object in that it is required to be published before being referenced. The EJB Home name specified in the EJB descriptor will be used to publish. For example:

```
deployejb -republish -temp temp -u scott -p tiger -encoding Unicode
          -s sess_iiop://local:2222:ORCL -descriptor Greeting.ejb server.jar
```

Because `deployejb` uses IIOP to connect to Oracle, the service name for the IIOP service of the database server has to be specified. Also, `server.jar` should contain the class files for the home interface object, remote interface object, and the bean implementation object of the EJB `Greeting`. Note that the `-encoding` argument is required if the EJB descriptor file `Greeting.ejb` is in different encoding from the default encoding of the Java VM. In this example, the `Greeting.ejb` is a Unicode text file.

EJB Client

An EJB client is like a CORBA client in that it can be a Java program using Oracle's JNDI interface to authenticate a session and look for the EJB object in the database server. To look for the corresponding EJB object, the EJB client looks for the home interface object whose name is specified in the EJB descriptor and calls the `create()` method of this home interface object to create the EJB instance in the database server. After the instance of the EJB is created, you can call the methods within it.

The following code shows how the EJB client calls the `Hello()` method of the EJB called `Demo.Greeting`. It is functionally equivalent to the code of the CORBA Client in the previous section, but uses the explicit authentication mechanism.

```
import Demo.Greeting;          //Remote interface object
import Demo.GreetingHome;      //Home interface object
import javax.naming.*;
import java.util.Hashtable;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.client.*;

public class Client
{
    public static void main (String[] args) throws Exception
    {
        Hashtable environment = new Hashtable ();
        environment.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        Context ic = new InitialContext (environment);
        // Login to the 9i server
        LoginServer lserver = (LoginServer)
            ic.lookup ("sess_iiop://local:2222:ORCL/etc/login");
```

```
Login li = new Login (lserver)
li.authenticate (username, password, null);
// Activate a Greeting instance in the 9i server
// This creates a first session in the server

GreetingHome greetingHome = (GreetingHome)
    ic.lookup ("sess_iiop://local:2222:ORCL/Demo/MyGreeting");
Greeting greet = greetingHome.create ();
System.out.println (greet.Hello (arg[0]));
}
}
```

Similar to the implicit authentication mechanism, the explicit authentication protocol, `li.authenticate()`, will automatically pass the default Java locale of the client to the `LoginServer` object in the database server. This Java locale will be used to initialize the Java locale of the server Java VM on which the EJB runs. In addition, the `NLS_LANGUAGE` and `NLS_TERRITORY` session parameters will be set to reflect this Java VM locale. This is to preserve the locale settings from EJB client to EJB server so that server uses the same language as the client.

Configurations for Multilingual Applications

To develop and deploy multilingual Java applications for Oracle9i, the database configurations and client environments for the targeted systems have to be determined.

Configuring a Multilingual Database

In order to store multilingual data into an Oracle9i database, you need to configure the database appropriately. There are two ways to store Unicode data into the database:

- As SQL `CHAR` datatypes in a Unicode database
- As SQL `NCHAR` datatypes in a non-Unicode database

See Also: [Chapter 5, "Supporting Multilingual Databases with Unicode"](#) for more information about choosing a Unicode solution and configuring the database for Unicode

Globalizing the Java Server Objects

For each Oracle9i session, a separate Java VM instance is created in the server for running the Java object, and Oracle9i Java support ensures that the locale of the Java VM instance is the same as that of the client Java VM. Hence the Java objects always run on the same locale in the database as the client locale.

For non-Java clients, the default locale of the Java VM instance will be the best matched Java locale corresponding to the `NLS_LANGUAGE` and `NLS_TERRITORY` session parameters propagated from the client `NLS_LANG` environment variable. In case of JSP and Java servlets, there is no `NLS_LANG` environment, the Java servlet is responsible to determine the locale of the client and synchronize it with the default Java locale of the Java VM instance on which the Java servlet runs.

Internationalizing the Java code

Java objects in the database such as Java stored procedures, Java servlets, Java CORBA, and EJB objects are server objects which are accessible from clients of different language preferences. They should be internationalized so that they are sensitive to the Java locale of the Java VM, which is initialized to the locale of the client.

With JDK internationalization support, you are able to specify a Java locale object to any locale-sensitive methods or use the default Java locale of the Java VM for those methods. Here are examples of how you may want to internationalize a Java stored procedure, Java servlet, Java CORBA object, or EJB:

- Externalize all localizable strings or objects from the Java code to resource bundles and make the resource bundles as part of the procedure, servlet, CORBA object, or EJB. Any messages returned from the resource bundle will be in the language of the client locale or whatever locale you specify.
- Use the Java formatting classes such as `DateFormat` and `NumberFormat` to format your date, time, numbers and currencies with the assumption that they will reflect the locale of the calling client.
- Use Java locale-sensitive string classes such as `Character`, `Collator`, and `BreakIterator` to check the classification of a character, compare two strings linguistically, and parse a string character by character.

Passing Multilingual Data Around

All Java server objects access the database with the JDBC Server-side Internal driver and should use either a Java string or `oracle.sql.CHAR` to represent string data to and from the database. Java strings are always encoded in UTF-16, and the

required conversion from the database character set to UTF-16 is transparently done as described previously. `oracle.sql.CHAR` stores the database data in byte array and tags it with a character set ID. It should be used when no string manipulation is required on the data. For example, `oracle.sql.CHAR` is the best choice for transferring string data from one table to another in the database.

When developing Java CORBA objects, the `wstring` data type should be used in the IDL as described in ["Java CORBA Object"](#) on page 9-24 to ensure that Unicode data is being passed from client to server.

Clients of Different Languages

Clients (or middle tiers) can have different language preferences, database access mechanisms, and Java runtime environments. The following are several commonly used client configurations.

- **Java CORBA clients running an ORB**

A CORBA client written in Java can access CORBA objects in the database server via IIOP. The client can be of different language environments. Upon login, the locale of the Java VM running the CORBA client will be automatically sent to the database ORB, and is used to initialize the Java VM session running the server objects. The use of the `wstring` data type of the server objects ensures the client and server communicate in Unicode.

- **Java applets running in browsers**

Java applets running in browsers can access the Oracle9i database via the JDBC Thin driver. No client-side Oracle library is required. The applets use the JDBC Thin driver to invoke SQL, PL/SQL as well as Java stored procedures. The JDBC Thin driver makes sure that Java stored procedures run in the same locale as the Java VM running the applets.

- **Dynamic HTML on browsers**

HTML pages invoke Java servlets via URLs over HTTP. The Java servlets running in the database construct dynamic HTML pages and deliver back to the browser. They should determine the locale of a user and construct the page according to the language and cultural convention preferences of the user.

- **Java applications running on client Java VMs**

Java applications running on the Java VM of the client machine can access the database via either JDBC OCI or JDBC Thin drivers. Java applications can also be a middle tier servlet running on a Web server. The applications use JDBC drivers to invoke SQL, PL/SQL as well as Java stored procedures. The JDBC

Thin and JDBC OCI drivers make sure that Java stored procedures will be running in the same locale as that of the client Java VM.

- C clients such as OCI, Pro*C, and ODBC
- Non-Java clients can call Java stored procedures the same way they call PL/SQL stored procedures. The Java VM locale is the best match of Oracle's language settings `NLS_LANGUAGE` and `NLS_TERRITORY` propagated from the `NLS_LANG` environment variable of the client. As a result, the client always gets messages from the server in the language specified by `NLS_LANG`. Data in the client are converted to and from the database character set by OCI.

Multilingual Demo Applications in SQLJ

This section contains a simple bookstore application written in SQLJ to demonstrate a database storing book information of different languages, and how SQLJ and JDBC are used to access the book information from the database. It also demonstrates the use of internationalized Java stored procedures to accomplish transactional tasks in the database server. The sample program consists of the following components:

- The SQLJ client Java application that displays a list of books in the store and allow users to add new books to and remove books from the inventory
- A Java stored procedure to add a new book to the inventory
- A Java stored procedure to remove an existing book from the inventory

The Database Schema

UTF8 is used as the database character set to store book information, such as names and authors, in languages around the world. The following tables in [Figure 9-7](#) are defined for storing the book and inventory information of the store.

Figure 9–7 Sample Tables

Book		Inventory	
Field Names	Data Types	Field Names	Data Types
ID (PRIMARY KEY)	NUMBER(10)	ID (PRIMARY KEY)	NUMBER(10)
NAME	VARCHAR(300)	LOCATION (PRIMARY KEY)	VARCHAR(90)
PUBLISH_DATE	DATE	QUANTITY	NUMBER(3)
AUTHOR	VARCHAR(120)		
PRICES	NUMBER(10,2)		

In addition, indexes are built with the NAME and AUTHOR columns of the BOOK table to speed up searching for books. A BOOKSEQ sequence will be created to generate a unique Book ID.

Java Stored Procedures

The Java class called Book is created to implement the methods Book.remove() and Book.add() that perform the tasks of removing books from and adding books to the inventory respectively. They are defined according to the following code. In this class, only the remove() method and the constructor are shown. The resource bundle BookRes.class is used to store localizable messages. The remove() method returns a message gotten from the resource bundle according to the current Java VM locale. There is no JDBC connection required to access the database because the stored procedure is already running in the context of a database session.

```
import java.sql.*;
import java.util.*;
import sqlj.runtime.ref.DefaultContext;
/* The book class implementation the transaction logics of the
   Java stored procedures.*/
public class Book
{
    static ResourceBundle rb;
    static int q, id;
    static DefaultContext ctx;
    public Book()
    {
        try
        {
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            DefaultContext.setDefaultContext(ctx);
```

```
        rb = java.util.ResourceBundle.getBundle("BookRes");
    }
    catch (Exception e)
    {
        System.out.println("Transaction failed: " + e.getMessage());
    }
}

public static String Remove(int id, int quantity, String location) throws
    SQLException
{
    rb = ResourceBundle.getBundle("BookRes");
    try
    {
        #sql {SELECT QUANTITY INTO :q FROM INVENTORY WHERE ID = :id AND
            LOCATION = :location};
        if (id == 1) return rb.getString ("NotEnough");
    }
    catch (Exception e)
    {
        return rb.getString ("NotEnough");
    }
    if ((q - quantity) == 0)
    {
        #sql {DELETE FROM INVENTORY WHERE ID = :id AND LOCATION = :location};
        try
        {
            #sql {SELECT SUM(QUANTITY) INTO :q FROM INVENTORY WHERE ID = :id};
        }
        catch (Exception e)
        {
            #sql { DELETE FROM BOOK WHERE ID = :id };
            return rb.getString("RemoveBook");
        }
        return rb.getString("RemoveInventory");
    }
    else
    {
        if ((q-quantity) < 0) return rb.getString ("NotEnough");
        #sql { UPDATE INVENTORY SET QUANTITY = :(q-quantity) WHERE ID = :id and
            LOCATION = :location };
        return rb.getString("DecreaseInventory");
    }
}

public static String Add( String bname, String author, String location,
    double price, int quantity, String publishdate ) throws SQLException
```

```
{
    rb = ResourceBundle.getBundle("BookRes");
    try
    {
        #sql { SELECT ID into :id FROM BOOK WHERE NAME = :bname AND AUTHOR =
        :author };
    }
    catch (Exception e)
    {
        #sql { SELECT BOOKSEQ.NEXTVAL INTO :id FROM DUAL };
        #sql { INSERT INTO BOOK VALUES (:id, :bname,
        TO_DATE(:publishdate,'YYYY-MM-DD'), :author, :price) };
        #sql { INSERT INTO INVENTORY VALUES (:id, :location, :quantity) };
        return rb.getString("AddBook");
    }
    try
    {
        #sql { SELECT QUANTITY INTO :q FROM INVENTORY WHERE ID = :id
        AND LOCATION = :location };
    }
    catch (Exception e)
    {
        #sql { INSERT INTO INVENTORY VALUES (:id, :location, :quantity) };
        return rb.getString("AddInventory");
    }
    #sql { UPDATE INVENTORY SET QUANTITY = :(q + quantity) WHERE ID = :id
    AND LOCATION = :location };
    return rb.getString("IncreaseInventory");
}
}
```

After the `Book.remove()` and `Book.add()` methods are defined, they are in turn published as Java stored functions in the database called `REMOVEBOOK()` and `ADDBOOK()` as follows:

```
CREATE FUNCTION REMOVEBOOK (ID NUMBER, QUANTITY NUMBER,
    LOCATION VARCHAR2)
    RETURN VARCHAR2
    AS LANGUAGE JAVA NAME
    'Book.remove(int, int, java.lang.String) return java.lang.String';

CREATE FUNCTION ADDBOOK (NAME VARCHAR2, AUTHOR VARCHAR2,
    LOCATION VARCHAR2, PRICE NUMBER, QUANTITY NUMBER, PUBLISH_DATE DATE)
    RETURN VARCHAR2
    AS LANGUAGE JAVA NAME
```



```
'Book.add(java.lang.String, java.lang.String, java.lang.String,  
double, int, java.sql.Date) return java.lang.String';
```

Note that the Java string returned will first be converted to a VARCHAR2 string, which is encoded in the database character set, before they are passed back to the client. If the database character is not UTF8, any Unicode characters in the Java strings that cannot be represented in the database character set will be replaced by ?. Similarly, the VARCHAR2 strings, which are encoded in the database character set, are converted to Java strings before being passed to the Java methods.

The SQLJ Client

The SQLJ client is a GUI Java application using either a JDBC Thin or JDBC OCI driver. It connects the client to a database, displays a list of books given a searching criterion, removes selected books from the inventory, and adds new books to the inventory. A class called `BookDB` is created to accomplish these tasks, and it is defined in the following code.

A `BookDB` object is created when the sample program starts up with the user name, password, and the location of the database. The methods are called from the GUI portion of the applications. The methods `removeBook()` and `addBook()` call the corresponding Java stored functions in the database and return the status of the transaction. The methods `searchByName()` and `searchByAuthor()` list books by name and author respectively, and store the results in the iterator `books` (the `BookRecs` class is generated by SQLJ) inside the `BookDB` object. The GUI code in turn calls the `getNextBook()` function to retrieve the list of books from the iterator object until a NULL is returned. The `getNextBook()` function simply fetches the next row from the iterator.

```
package sqlj.bookstore;  
  
import java.sql.*;  
import sqlj.bookstore.BookDescription;  
import sqlj.runtime.ref.DefaultContext;  
import java.util.Locale;  
/*The iterator used for a book description when communicating with the server*/  
#sql iterator BooksRecs( int ID, String NAME, String AUTHOR, Date PUBLISH_DATE,  
                        String LOCATION, int QUANTITY, double PRICE);  
/*This is the class used for connection to the server.*/  
class BookDB  
{  
    static public final String DRIVER = "oracle.jdbc.driver.OracleDriver";  
    static public final String URL_PREFIX = "jdbc:oracle:thin:@";  
    private DefaultContext m_ctx = null;
```

```
private String msg;
private BooksRecs books;
/*Constructor - registers the driver*/
BookDb()
{
    try
    {
        DriverManager.registerDriver
            ((Driver) (Class.forName(DRIVER).newInstance()));
    }
    catch (Exception e)
    {
        System.exit(1);
    }
}
/*Connect to the database.*/
DefaultContext connect(String id, String pwd, String userUrl) throws
SQLException
{
    String url = new String(URL_PREFIX);
    url = url.concat(userUrl);
    Connection conn = null;
    if (m_ctx != null) return m_ctx;
    try
    {
        conn = DriverManager.getConnection(url, id, pwd);
    }
    catch (SQLException e)
    {
        throw(e);
    }
    if (m_ctx == null)
    {
        try
        {
            m_ctx = new DefaultContext(conn);
        }
        catch (SQLException e)
        {
            throw(e);
        }
    }
    return m_ctx;
}
/*Add a new book to the database.*/
```

```
public String addBook(BookDescription book)
{
    String name = book.getTitle();
    String author = book.getAuthor();
    String date = book.getPublishDateString();
    String location = book.getLocation();
    int quantity = book.getQuantity();
    double price = book.getPrice();
    try
    {
        #sql [m_ctx] msg = {VALUE ( ADDBOOK ( :name, :author, :location,
        :price, :quantity, :date))};
        #sql [m_ctx] {COMMIT};
    }
    catch (SQLException e)
    {
        return (e.getMessage());
    }
    return msg;
}
/*Remove a book.*/
public String removeBook(int id, int quantity, String location)
{
    try
    {
        #sql [m_ctx] msg = {VALUE ( REMOVEBOOK ( :id, :quantity,
        :location))};
        #sql [m_ctx] {COMMIT};
    }
    catch (SQLException e)
    {
        return (e.getMessage());
    }
    return msg;
}
/*Search books by the given author.*/
public void searchByAuthor(String author)
{
    String key = "%" + author + "%";
    books = null;
    System.gc();
    try
    {
        #sql [m_ctx] books = { SELECT BOOK.ID, NAME, AUTHOR, PUBLISH_DATE,
        LOCATION, QUANTITY, PRICE
```

```
        FROM BOOK, INVENTORY WHERE BOOK.ID = INVENTORY.ID AND AUTHOR LIKE
        :key ORDER BY BOOK.ID};
    }
    catch (SQLException e) {}
}
/*Search books with the given title.*/
public void searchByTitle(String title)
{
    String key = "%" + title + "%";
    books = null;
    System.gc();
    try
    {
        #sql [m_ctx] books = { SELECT BOOK.ID, NAME, AUTHOR, PUBLISH_DATE,
        LOCATION, QUANTITY, PRICE
        FROM BOOK, INVENTORY WHERE BOOK.ID = INVENTORY.ID AND NAME LIKE
        :key ORDER BY BOOK.ID};
    }
    catch (SQLException e) {}
}
/*Returns the next BookDescription from the last search, null if at the
end of the result list.*/
public BookDescription getNextBook()
{
    BookDescription book = null;
    try
    {
        if (books.next())
        {
            book = new BookDescription(books.ID(), books.AUTHOR(), books.NAME(),
            books.PUBLISH_DATE(), books.PRICE(),
            books.LOCATION(), books.QUANTITY());
        }
    }
    catch (SQLException e) {}
    return book;
}
}
```

Character Set Scanner Utility

This chapter introduces the Character Set Scanner Utility, a National Language Support utility for checking data before migrating character sets. The topics in this chapter include:

- [Overview of Choosing and Migrating Character Sets](#)
- [Database Character Set Migration](#)
- [What is the Character Set Scanner Utility?](#)
- [Scan Modes in the Scanner](#)
- [Using The Scanner](#)
- [Scanner Parameters](#)
- [Sample Scanner Sessions](#)
- [Storage and Performance Considerations in the Scanner](#)
- [Scanner Utility Reference Material](#)

Overview of Choosing and Migrating Character Sets

Choosing the appropriate database character set for your database is an important decision and requires taking into account many factors. Some of these factors are:

- The type of data you need to store
- The number of languages the database character set can represent
- The different sizing requirements of each character set and their performance implications

A related topic is choosing a new character set for an existing database, which is called migrating character sets. Migrating from one database character set to another involves additional considerations beyond those of simply choosing a character set. In particular, it is a complex planning process with the goal of minimizing the possibility of losing data because of data truncation and character set conversions during the migration.

See Also: [Chapter 3, "Setting Up a Globalization Support Environment"](#) for more information about choosing character sets

Data Truncation

When the database is created using byte semantics, the sizes of character datatypes `CHAR` and `VARCHAR2` are specified in bytes, not characters. Hence, the specification `CHAR(20)` in a table definition allows 20 bytes for storing character data. This is acceptable when the database character set uses a single-byte character encoding scheme because the number of characters will be equivalent to the number of bytes. If the database character set uses a multibyte character encoding scheme, however, there is no such correspondence. That is, the number of bytes no longer equals the number of characters because a character can consist of one or more bytes. This situation can cause problems.

During migration to a new character set, it is important to verify the column widths of existing `CHAR` and `VARCHAR` columns because they might need to be extended to support encoding that requires multibyte storage. If the character set width differs during the import process, truncation of data can occur if conversion causes expansion of data. [Figure 10–1](#) shows a typical case of data expansion with single-byte characters becoming multibyte. In it, ä (a with an umlaut) is a single-byte character in `WE8MSWIN1252`, but it becomes a double-byte character in `UTF8`. Also, the Euro symbol goes from one byte to three bytes in this conversion.

Figure 10–1 Single-byte Character Sets Becoming Multibyte

Char	WE8MSWIN1252	UTF8
ä	E4	C3 A4
ö	F6	C3 B6
©	A9	C2 A9
€	80	E2 82 AC

The maximum number of bytes for `CHAR` and `VARCHAR2` data types are 2000 and 4000 respectively. If the data columns in the new destination character set require more than 2000 and 4000 bytes, you need to change your schema.

Restrictions

The following are some known restrictions caused by data truncation:

- Within the database data dictionary, schema object names cannot exceed 30 bytes in length. Schema objects are tables, clusters, views, indexes, synonyms, tablespaces, and usernames. Renaming schema objects is required if they exceed 30 bytes in the new database character set. For example, one Thai character in the Thai national character set requires 1 byte, but, in UTF8, it requires 3 bytes. So, if you have defined a table with 11 Thai characters, then this table name must be shortened to 10 or fewer Thai characters when changing the database character set to UTF8.
- If your existing Oracle usernames or passwords are created based on characters that will change in size in the target character set, these users will experience login difficulties due to authentication failures after the migration to a new character set. This is because the encrypted usernames and passwords stored in the data dictionary are not updated during migration to a new character set. For example, assuming the current database character set is WE8MSWIN1252 and the target database character set is UTF8, the username `scött` (o with an umlaut) will change from 5 bytes in WE8MSWIN1252 to 6 bytes. In UTF8, `scött` will no longer be able to log in because of the difference in the length of the username. Oracle recommends that usernames and passwords be based on ASCII characters. If they are not, you will need to reset the affected usernames and passwords after migrating to a new character set.

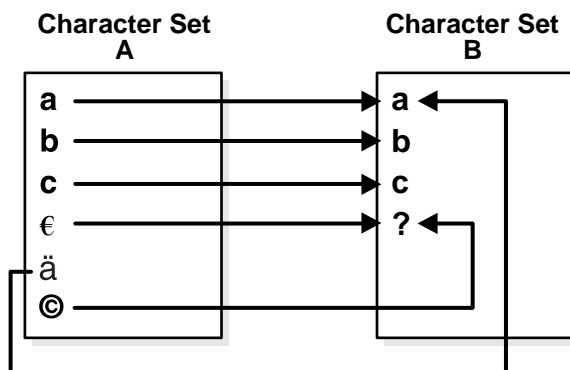
- When CHAR data contains characters that will be expanded after migration to a new character set, space padding will not be removed during database export by default. This means that these rows will be rejected upon import into the database with the new character set. The workaround is to set the `BLANK_TRIMMING` initialization parameter to `TRUE` prior to the import.

See Also: *Oracle9i Database Reference* for more information about `BLANK_TRIMMING`

Character Set Conversions

When migrating to a new database character set, the Export and Import utilities can handle character set conversions from the original database character set to the new database character set. However, character set conversions can sometimes cause data loss or data corruption. For example, if you are migrating from character set A to character set B, the destination character set B should be a superset of character set A. Characters that are not available in character set B will be converted to replacement characters, which are usually specified as `?` or `?` or other linguistically-related characters. For example, `ä` (a with an umlaut) can be converted to `a`. Replacement characters are defined by the target character set. [Figure 10-2](#) shows a sample conversion where the copyright and Euro symbols are converted to `?` and `ä` to `a`.

Figure 10-2 Replacement Characters in Character Set Conversion

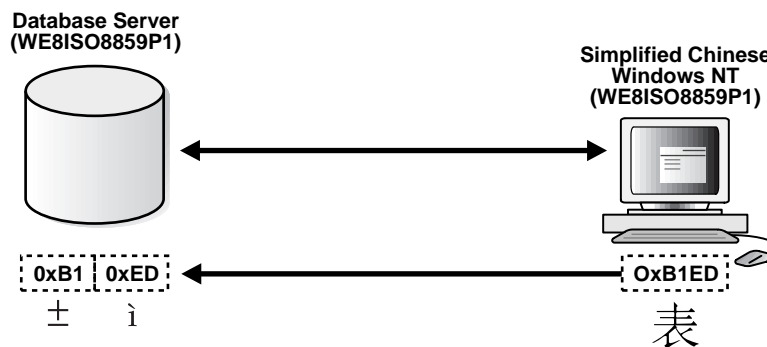


To reduce the risk of losing data, choose a destination character set with similar character repertoires, if possible. Migrating to Unicode can be an attractive option because UTF8 contains characters from most legacy character sets.

Another scenario that can cause the loss of data is migrating a database containing data of a different character set from that of the database character set. Users can insert data into the database from another character set if the client `NLS_LANG` character set setting is the same as the database character set. When these settings are the same, Oracle assumes that the data being sent or received is from the same character set, so no validations or conversions are performed.

This can lead to two possible data inconsistency problems. One problem occurs when a database contains data from another character set but the same codepoints exist in both character sets. For example, if the database character set is `WE8ISO8859P1` and the end user Chinese Windows NT client's `NLS_LANG` setting is `SIMPLIFIED CHINESE_CHINA.WE8ISO8859P1`, then all multibyte Chinese data (from the `ZHS16GBK` character set) is stored as multiples of single-byte `WE8ISO8859P1` data. This means that Oracle will treat these characters as single-byte `WE8ISO8859P1` characters. Hence all SQL string manipulation functions such as `SUBSTR` or `LENGTH` will be based on bytes rather than characters. All bytes constituting `ZHS16GBK` data are legal `WE8ISO8859P1` codes. If such a database is migrated to another character set, for example, `UTF8`, character codes will be converted as if they were in `WE8ISO8859P1`. This way, each of the two bytes of a `ZHS16GBK` character will be converted separately, yielding meaningless values in `UTF8`. [Figure 10-3](#) shows an example of this incorrect character set replacement.

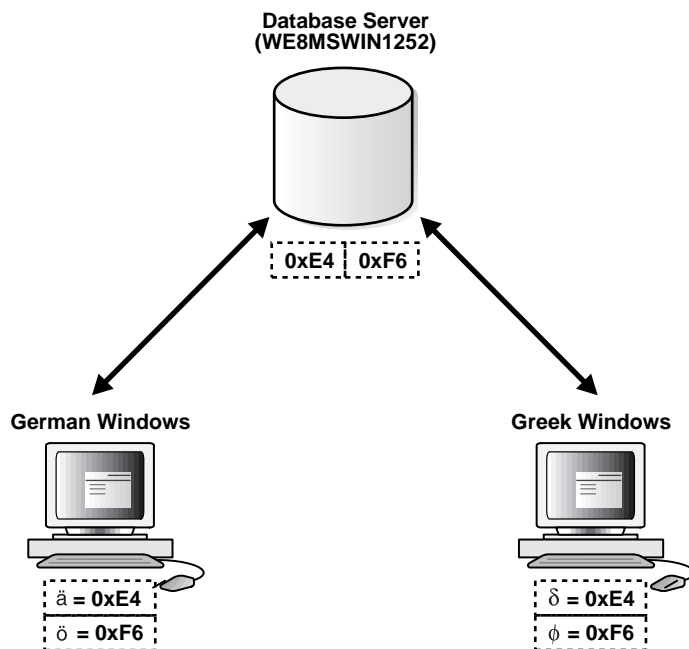
Figure 10-3 Incorrect Character Set Replacement



The second possibility is having data from mixed character sets inside the database. For example, if the data character set is `WE8MSWIN1252`, and two separate Windows clients using German and Greek are both using the `NLS_LANG` character set setting as `WE8MSWIN1252`, then the database will contain a mixture of German

and Greek characters. [Figure 10–4](#) shows how different clients can use different character sets in the same database.

Figure 10–4 *Mixed Character Sets*



For database character set migration to be successful, both of these cases require manual intervention because Oracle cannot determine the character sets of the data being stored.

Database Character Set Migration

Database character set migration has two distinct stages:

- [Data Scanning](#)
- [Conversion of Data](#)

Data Scanning

Before you actually migrate your character set, you need to identify areas of possible database character set conversions and truncation of data. This step is called **data scanning**.

Data scanning identifies the amount of effort required to migrate data into the new character encoding scheme prior to the change of the database character set. Some examples of what are found during a data scan are the number of schema objects where the column widths need to be expanded and the extent of the data that does not exist in the target repertoire. This information will assist in determining the best approach for the conversion of the database character set.

Conversion of Data

There are generally three approaches in migrating data from one database character set to another, if the database does not contain any of the inconsistencies described in "[Character Set Conversions](#)" on page 10-4. A description of methods to migrate databases with such inconsistencies is out of the scope of this document. For more information, contact Oracle Consulting Services for assistance.

In most cases, a full export or import is recommended to properly convert all data to a new character set. It is important to be aware of data truncation issues because character data type columns might need to be extended prior to import to handle the increase in size required. Existing PL/SQL code should be reviewed to ensure all byte-based SQL functions such as `LENGTHB`, `SUBSTRB`, and `INSTRB`, and PL/SQL `CHAR` and `VARCHAR2` declarations are still valid. However, if, and only if, the new character set is a strict superset of the current character set, you can use the `ALTER DATABASE CHARACTER SET` statement to expedite migration to a new database character set. The target character set is a strict superset if, and only if, each and every character in the source character set is available in the target character set with the same corresponding codepoint value. For instance, because `US7ASCII` is a strict subset of `UTF8`, then an `ALTER DATABASE CHARACTER SET` statement can be used to upgrade the database character set from `US7ASCII` to `UTF8`.

See Also: [Appendix A, "Locale Data"](#) for a listing of all superset character sets

ALTER DATABASE CHARACTER SET Statement Restrictions

In Oracle9i, `CLOB` data is stored as UCS-2 (2-byte fixed-width Unicode) for multibyte database character sets. For single-byte database character sets, `CLOB`

data is stored as the database character set. Because the `ALTER DATABASE CHARACTER SET` statement does not perform any data conversion, if the database character set is migrated from single-byte to multiple byte using ADCS, then `CLOB` columns will remain in the original database character set encoding. This introduces data inconsistency in the `CLOB` columns. Likewise, if you migrate from one Unicode national character set to another, the `SQL NCHAR` datatype columns will be corrupted.

The migration procedure for `CLOB` and the `SQL NCHAR` datatype columns is:

1. Export the tables containing `CLOB` and `SQL NCHAR` columns.
2. Drop the tables containing `CLOB` and `SQL NCHAR` columns.
3. Use the `ALTER DATABASE CHARACTER SET` and `ALTER DATABASE NATIONAL CHARACTER SET` statements.
4. Import.

The syntax is:

```
ALTER DATABASE [db_name] CHARACTER SET new_character_set;  
ALTER DATABASE [db_name] NATIONAL CHARACTER SET new_NCHAR_character_set;
```

The database name is optional. The character set name should be specified without quotes. For example:

```
ALTER DATABASE CHARACTER SET UTF8;
```

To change the database character set, perform the following steps:

1. Shut down the database, using either a `SHUTDOWN IMMEDIATE` or a `SHUTDOWN NORMAL` statement.
2. Do a full backup.
3. Complete the following statements:

```
STARTUP MOUNT;  
ALTER SYSTEM ENABLE RESTRICTED SESSION;  
ALTER SYSTEM SET JOB_QUEUE_PROCESSES=0;  
ALTER SYSTEM SET AQ_TM_PROCESSES=0;  
ALTER DATABASE OPEN;  
ALTER DATABASE CHARACTER SET <new_character_set_name>;  
SHUTDOWN IMMEDIATE;  -- or NORMAL  
STARTUP;
```

To change the national character set, replace the `ALTER DATABASE CHARACTER SET` statement with the `ALTER DATABASE NATIONAL CHARACTER SET` statement. You can issue both statements together if desired.

See Also: *Oracle9i SQL Reference* for the syntax of the `ALTER DATABASE [NATIONAL] CHARACTER SET` statement

When using Oracle9i Real Application Clusters, ensure that no other Oracle background processes are running, with the exception of the one session through which a user is connected, before attempting to issue the `ALTER DATABASE CHARACTER SET` statement. Use the following SQL statement to verify your environment:

```
SELECT SID, SERIAL#, PROGRAM FROM V$SESSION;
```

Setting the initialization parameter `PARALLEL_SERVER` to `FALSE` allows the character set change to go through. This is required in an Oracle9i Real Application Cluster environment; an exclusive startup is not sufficient.

Note: It is essential to do a full backup of the database before using the `ALTER DATABASE [NATIONAL] CHARACTER SET` statement because the command cannot be rolled back.

The last approach is to perform an `ALTER DATABASE CHARACTER SET` statement followed by selective imports. This method is best suited for a known distribution of convertible data that is stored within a small number of tables. A full export and import will be too expensive in this scenario. For example, suppose you have a 100GB database with over 300 tables, but only 3 tables requires character set conversions. The rest of the data is of the same encoding as the destination character set. The 3 tables can be exported and imported back to the new database after issuing the `ALTER DATABASE CHARACTER SET` statement.

Incorrect data conversion can lead to data corruption, so perform a full backup of the database before attempting to migrate the data to a new character set.

What is the Character Set Scanner Utility?

The Character Set Scanner provides an assessment of the feasibility and potential issues in migrating an Oracle database to a new database character set. The Scanner checks all character data in the database and tests for the effects and problems of changing the character set encoding. At the end of the scan, it generates a summary

report of the database scan. This report provides estimates of the amount of work required to convert the database to a new character set.

Based on the information in the summary report, you will be able to decide on the most appropriate method to migrate the database's character set. The methods are:

- Export and Import utilities
- ALTER DATABASE CHARACTER SET statement
- ALTER DATABASE CHARACTER SET with selective Export and Import

Note: If there are conversion exceptions reported by the Scanner, these problems must be fixed first before using any of the above methods to do the conversions. This may involve modifying the problem data to eliminate those exceptions. In extreme cases, both database and application might need to be modified. Oracle Corporation recommends you contact Oracle Consulting Services for services on database character set migration.

Conversion Tests on Character Data

The Scanner reads the character data and tests for the following conditions on each data cell:

- Do character codes of the data cells change when converted to the new character set?
- Can the data cells be successfully converted to the new character set?
- Will the post-conversion data fit into the current column size?

The Scanner reads and tests for data in CHAR, VARCHAR2, LONG, CLOB, NCHAR, NVARCHAR2, and NCLOB columns only. The Scanner does not perform post-conversion column size testing for LONG, CLOB, and NCLOB columns.

Access Privileges

To use the Scanner, you must have DBA privileges on the Oracle database.

Restrictions

All the character-based data in CHAR, VARCHAR2, LONG, and CLOB columns is stored in the same character set, which is the database character set specified with the CREATE DATABASE statement when the database was first created. However, in

some configurations, it is possible to store data in a different character set from the database character set either intentionally or unintentionally. This happens most often when the `NLS_LANG` character set is the same as the database character set, because in such cases Oracle sends and receives data as is, without any conversion or validation. But it can also happen if one of the two character sets is a superset of the other, in which case many of the codes appear as if they were not converted. For example, if `NLS_LANG` is set to `WE8ISO8859P1` and the database character set is `WE8MSWIN1252`, all codes except the range 128-159 are preserved through the client/server conversion.

Although a database that contains data not in its database character set cannot be converted to another character set by the three methods described in "[What is the Character Set Scanner Utility?](#)" on page 10-9, you can still use the Scanner in the way described below to test the effect of the conversion that would take place if the data were in the database character set.

Database Containing Data From Two or More Character Sets

If a database contains data from more than one character set, the Scanner cannot accurately test the effects of changing the database character set on the database because it cannot differentiate character sets properly. If the data can be divided into two separate tables, one for each language, then the Scanner can perform two single table scans to verify the validity of the data.

For each scan, a different value of the `FROMCHAR` parameter can be used to tell the Scanner to treat all `CHAR`, `VARCHAR2`, `LONG`, and `CLOB` columns in the table as if they were in the specified character set.

Database Containing Data Not From the Database Character Set

If a database contains data not in the database character set, but still in only one character set, the Scanner can perform a full database scan. Use the `FROMCHAR` parameter to tell the Scanner what character set the data is in.

Scan Modes in the Scanner

The Character Set Scanner provides three modes of database scan:

- [Full Database Scan](#)
- [User Tables Scan](#)
- [Single Table Scan](#)

Full Database Scan

The Scanner reads and verifies the character data of all tables belonging to all users in the database including the data dictionary (*SYS* user), and it reports on the effects of the simulated migration to the new database character set. It scans all schema objects including stored packages, procedures and functions, and object names.

To understand the feasibility of migration to a new database character set, you need to perform a full database scan.

User Tables Scan

The Scanner reads and verifies character data of all tables belonging to the specified user and reports on the effects of changing the character set on them.

The Scanner does not test for table definitions such as table names and column names. To see the effects on the schema definitions, you need to perform a full database scan.

Single Table Scan

The Scanner reads and verifies the character data of the specified table, and reports the effects on changing the character set of them.

The Scanner does not test for table definitions such as table name and column name. To see the effects on the schema definitions, you need to perform a full database scan.

Using The Scanner

This section describes how to use the Scanner, including the steps you need to perform before scanning and the procedures on how to invoke the Scanner. The topics discussed are:

- [Before Using the Scanner](#)
- [Compatibility](#)
- [Invoking the Scanner](#)
- [Getting Online Help for the Scanner](#)
- [The Parameter File](#)

Before Using the Scanner

To use the Scanner, you must run the `CSMINST.SQL` script on the database that you plan to scan. `CSMINST.SQL` needs to be run only once, so it is not necessary to run it each time you scan the database. The script performs the following tasks to prepare the database for scanning:

- Creates a user named `CSMIG`
- Assigns the necessary privileges to `CSMIG`
- Assigns the default tablespace to `CSMIG`
- Connects as `CSMIG`
- Creates the Scanner system tables under `CSMIG`

The `SYSTEM` tablespace is assigned to `CSMIG` by default, so you need to ensure there is sufficient storage space available in the `SYSTEM` tablespace before scanning the database. The amount of space required depends on the type of scan and the nature of the data in the database.

See Also: ["Storage and Performance Considerations in the Scanner"](#) on page 10-35

You can modify the default tablespace for `CSMIG` by editing the script `CSMINST.SQL`. Modify the following statement in `CSMINST.SQL` to assign your preferred tablespace to `CSMIG` as follows:

```
ALTER USER CSMIG default tablespace PREFERRED_TABLESPACE;
```

Then run `CSMINST.SQL` using these commands and SQL statements:

```
% cd $ORACLE_HOME/rdbms/admin
% sqlplus
SQL> CONNECT system/manager as sysdba
SQL> START csmnst.sql
```

Compatibility

The Scanner is certified with Oracle databases on any platforms running under the same release except you cannot mix ASCII- and EBCDIC-based platforms. For example, the release 9.0.1 versions of the Scanner on any ASCII-based client platforms are certified to run with any release 9.0.1 Oracle databases on any ASCII-based platforms, while EBCDIC-based clients are certified to run with any release 9.0.1 Oracle database on EBCDIC platforms.

Oracle Corporation recommends that you run the Scanner in the same Oracle Home as the database when possible.

Invoking the Scanner

You can invoke the Scanner by one of these methods:

- Using the parameter file

```
csscan system/manager PARFILE=filename
```

PARFILE is a file containing the Scanner parameters you typically use.

- Using the command line

```
csscan system/manager full=y tochar=utf8 array=10240 process=3
```

- Using an interactive session

```
csscan system/manager
```

In an interactive session, the Scanner prompts you for the following parameters:

- USERID
- FULL
- USER
- TABLE
- TOCHAR
- ARRAY
- PROCESS

If you want to specify parameters that are not listed above, you need to invoke the Scanner using either the parameter file or the command line.

Getting Online Help for the Scanner

The Scanner provides online help. Enter `csscan help=y` on the command line to invoke the help screen.

```
Character Set Scanner: Release 9.0.1.0.0 - Production
```

```
(c) Copyright 2001 Oracle Corporation. All rights reserved.
```

You can let Scanner prompt you for parameters by entering the `CSSCAN` command followed by your username and password. For example:

```
CSSCAN SYSTEM/MANAGER
```

Alternatively, you can control how Scanner runs by entering the `CSSCAN` command followed by various parameters. To specify parameters, use keywords. For example:

```
CSSCAN SYSTEM/MANAGER FULL=y TOCHAR=utf8 ARRAY=102400 PROCESS=3
```

The following is a list of Scanner keywords:

Keyword	Default	Prompt	Description
-----			-----
USERID		yes	username/password
FULL	N	yes	scan entire database
USER		yes	user name of the table to scan
TABLE		yes	table name to scan
TOCHAR		yes	new database character set name
FROMCHAR			current database character set name
TONCHAR			new NCHAR character set name
FROMNCHAR			current NCHAR character set name
ARRAY	10240	yes	size of array fetch buffer
PROCESS	1	yes	number of scan process
MAXBLOCKS			split table if larger than MAXBLOCKS
CAPTURE	N		capture convertible data
SUPPRESS			suppress error log by N per table
FEEDBACK			feedback progress every N rows
BOUNDARIES			list of column size boundaries for summary report
LASTRPT	N		generate report of the last database scan
LOG	scan		base name of log files
PARFILE			parameter file name
HELP	N		show help screen (this screen)

The Parameter File

The parameter file allows you to specify Scanner parameters in a file where they can be easily modified or reused. Create a parameter file using any flat file text editor.

The command line option `PARFILE=filename` tells the Scanner to read the parameters from a specified file rather than from the command line. For example:

```
csscan parfile=filename
```

or

```
csscan username/password parfile=filename
```

The syntax for parameter file specifications is one of the following:

```
KEYWORD=value  
KEYWORD=(value1, value2, ...)
```

The following is an example of a parameter file:

```
USERID=system/manager  
USER=SCOTT      # scan SCOTT's tables  
TOCHAR=utf8  
ARRAY=40960  
PROCESS=2      # use two concurrent scan processes  
FEEDBACK=1000
```

You can add comments to the parameter file by preceding them with the pound (#) sign. All characters to the right of the pound sign are ignored.

Scanner Parameters

This section describes each of the Scanner parameters.

ARRAY

Default value:	10240
Minimum value:	4096
Maximum value:	unlimited
Purpose:	Specifies the size in bytes of the array buffer used to fetch data. The size of the array buffer determines the number of rows fetched by the Scanner at any one time.

The formula below gives an approximation of number of rows fetched at a time:

```
(rows in array) =  
(ARRAY buffer size) / (sum of the CHAR and VARCHAR2 column sizes of a given table)
```

If the summation of CHAR and VARCHAR2 column sizes exceeds the array buffer size, then the Scanner fetches only one row at a time. Tables with LONG, CLOB, or NCLOB columns are fetched only one row at a time.

This parameter affects the duration of a database scan. In general, the larger the size of the array buffer, the shorter the duration time. Each scan process will allocate the specified size of array buffer.

BOUNDARIES

Default value:	none
Purpose:	Specifies the list of column boundary sizes that are used for an application data conversion summary report. This parameter is used to locate the distribution of the application data for the datatypes CHAR, VARCHAR2, NCHAR, and NVARCHAR2.

For example, if you specify a BOUNDARIES value of (10, 100, 1000), the application data conversion summary report will produce a breakdown of the CHAR data into the following groups by their column length, CHAR(1..10), CHAR(11..100) and CHAR(101..1000), likewise for the VARCHAR2, NCHAR, and NVARCHAR2 datatypes.

CAPTURE

Default value:	N
Range of values:	Y or N
Purpose:	Indicates whether to capture the information on the individual convertible rows as well as the default of storing the exception rows. The convertible rows information is written to the table CSM\$ERRORS if the parameter CAPTURE is set to Y. This information can be used to deduce which records need to be converted to the target character set by selective export and import.

FEEDBACK

Default value:	none
Minimum value:	100
Maximum value:	100000

Purpose:	Specifies that the Scanner should display a progress meter in the form of a dot for every N number of rows scanned.
-----------------	---

For example, if you specify `FEEDBACK=1000`, the Scanner displays a dot for every 1000 rows scanned. The `FEEDBACK` value applies to all tables being scanned, so it cannot be set on a per-table basis.

FROMCHAR

Default value:	none
Purpose:	Specifies the actual character set name for <code>CHAR</code> , <code>VARCHAR2</code> , <code>LONG</code> , and <code>CLOB</code> data types in the database. By default, the Scanner assumes the character set for the above data types to be the database character set.

Use this parameter to override the default database character set definition for `CHAR`, `VARCHAR2`, `LONG`, and `CLOB` data in the database.

FROMNCHAR

Default value:	none
Purpose:	Specifies the actual national database character set name for <code>NCHAR</code> , <code>NVARCHAR2</code> , and <code>NCLOB</code> data types in the database. By default, the Scanner assumes the character set for the above data types to be the database national character set.

Use this parameter to override the default database character set definition for `NCHAR`, `NVARCHAR2`, and `NCLOB` data in the database.

FULL

Default value:	N
Range of values:	Y or N

Purpose:	Indicates whether to perform the full database scan (that is, to scan the entire database including the data dictionary). Specify <code>FULL=Y</code> to scan in full database mode.
-----------------	--

For more information on full database scans, refer to "[Scan Modes in the Scanner](#)" on page 10-11.

HELP

Default value:	N
Range of values:	Y or N
Purpose:	Displays a help message with descriptions of the Scanner parameters.

For more information, see "[Getting Online Help for the Scanner](#)" on page 10-14.

LASTRPT

Default value:	N
Range of values:	Y or N
Purpose:	Indicates whether to regenerate the Scanner reports based on statistics gathered from the last database scan.

If `LASTRPT=Y` is specified, the Scanner does not scan the database, but creates the report files using the information left by the previous database scan session instead.

If `LASTRPT=Y` is specified, only the `USERID`, `BOUNDARIES`, and `LOG` parameters take effect.

LOG

Default value:	scan
-----------------------	------

Purpose:	Specifies a base file name for the following Scanner report files: Database Scan Summary Report file whose extension is <code>.txt</code> Individual Exception Report file whose extension is <code>.err</code> Screen log file whose extension is <code>.out</code>
-----------------	---

By default, the Scanner generates the three text files, `scan.txt`, `scan.err`, and `scan.out` in the current directory.

MAXBLOCKS

Default value:	none
Minimum value:	1000
Maximum value:	unlimited
Purpose:	Specifies the maximum block size per table, so that large tables can be split into smaller chunks for the Scanner to process.

For example, if the `MAXBLOCKS` parameter is set to 1000, then any tables that are greater than 1000 blocks in size will be divided into `n` chunks, where `n=CEIL(table block size/1000)`.

Dividing large tables into smaller pieces will be beneficial only when the number of processes set with `PROCESS` is greater than 1. If the `MAXBLOCKS` parameter is not set, the Scanner attempts to split up large tables based on the its own optimization rules.

PARFILE

Default value:	none
Purpose:	Specifies a filename for a file that contains a list of Scanner parameters.

See Also: ["The Parameter File"](#) on page 10-15

PROCESS

Default value:	1
Minimum value:	1
Maximum value:	32
Purpose:	Specifies the number of concurrent scan processes to utilize for the database scan.

SUPPRESS

Default value:	unlimited
Minimum value:	0
Maximum value:	unlimited
Purpose:	Specifies the maximum number of data exceptions being logged per table.

The Scanner inserts individual exceptional record information into the CSM\$ERRORS table when an exception is found in a data cell. The table grows depending on the number of exceptions reported.

This parameter is used to suppress the logging of individual exception information after a specified number of exceptions are inserted per table. For example, if SUPPRESS is set to 100, then the Scanner records a maximum of 100 exception records per table.

See Also: ["Storage Considerations"](#) on page 10-35

TABLE

Default value:	none
Purpose:	Specifies the name of the table to scan.

When specified, Scanner scans the specified table only. For example, the command below scans the emp table that belongs to the user scott:

```
csscan system/manager USER=SCOTT TABLE=EMP ...
```

TOCHAR

Default value:	none
Purpose:	Specifies a target database character set name for the CHAR, VARCHAR2, LONG, and CLOB data.

TONCHAR

Default value:	none
Purpose:	Specifies a target database character set name for the NCHAR, NVARCHAR2, and NCLOB data.

If you do not specify a value for TONCHAR, then the Scanner does not scan NCHAR, NVARCHAR2, and NCLOB data.

USER

Default value:	none
Purpose:	Specifies the owner of the tables to be scanned.

If the parameter USER is specified, then the Scanner scans all tables belonging to the user. If TABLE is specified, the Scanner scans only the table specified by TABLE that belongs to the user. For example, the following statement scans all tables belonging to the user scott:

```
csscan system/manager USER=scott ...
```

USERID

Default value:	none
-----------------------	------

Purpose:	Specifies the username and password (and optional connect string) of the user who scans the database. If you omit the password, then the Scanner prompts you for it.
-----------------	--

The following examples are all valid:

```
username/password
username/password@connect_string
username
username@connect_string
```

Sample Scanner Sessions

The following examples show you how to use the command line and parameter file methods to use Full Database, User Tables, and Single Table scan modes.

Sample Session of Full Database Scan

The following example shows how to scan the full database to see the effects on migrating it to UTF8. This example assumes the current database character set is WE8ISO8859P1 (or anything other than UTF8).

Parameter File Method

```
% csscan system/manager parfile=param.txt
```

The param.txt file contains the following information:

```
full=y
tochar=utf8
array=40960
process=4
```

Command Line Method

```
% csscan system/manager full=y tochar=utf8 array=40960 process=4
```

Scanner Messages

Database Scanner: Release 9.0.1.0.0 - Production

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

Connected to:

Oracle9 Enterprise Edition Release 9.0.1.0.0 - Production

With the Objects option
PL/SQL Release 9.0.1.0.0 - Production

Enumerating tables to scan...

```
. process 1 scanning SYSTEM.REPCAT$_RESOLUTION
. process 1 scanning SYS.AQ$_MESSAGE_TYPES
. process 1 scanning SYS.ARGUMENT$
. process 2 scanning SYS.AUD$
. process 3 scanning SYS.ATTRIBUTE$
. process 4 scanning SYS.ATTRCOL$
. process 2 scanning SYS.AUDIT_ACTIONS
. process 2 scanning SYS.Bootstrap$
. process 2 scanning SYS.CCOL$
. process 2 scanning SYS.CDEF$
:
:
. process 3 scanning SYSTEM.REPCAT$_REPOBJECT
. process 1 scanning SYSTEM.REPCAT$_REPPROP
. process 2 scanning SYSTEM.REPCAT$_REPSchema
. process 3 scanning MDSYS.MD$DIM
. process 1 scanning MDSYS.MD$DICTVER
. process 2 scanning MDSYS.MD$EXC
. process 3 scanning MDSYS.MD$LER
. process 1 scanning MDSYS.MD$PTAB
. process 2 scanning MDSYS.MD$PTS
. process 3 scanning MDSYS.MD$TAB
```

Creating Database Scan Summary Report...

Creating Individual Exception Report...

Scanner terminated successfully.

Sample Session of User Tables Scan

The following example shows how to scan the user tables to see the effects on migrating them to UTF8. This example assumes the current database character set is US7ASCII, but the actual data stored is in Western European WE8MSWIN1252 encoding.

Parameter File Method

```
% csscan system/manager parfile=param.txt
```

The param.txt file contains the following information:

```
user=scott
fromchar=we8mswin1252
tochar=utf8
array=40960
process=1
```

Command Line Method

```
% csscan system/manager user=scott fromchar=we8mswin1252 tochar=utf8 array=40960
process=1
```

Scanner Messages

Database Scanner: Release 9.0.1.0.0 - Production

(c) Copyright 2000 Oracle Corporation. All rights reserved.

```
Connected to:
Oracle8 Enterprise Edition Release 9.0.1.0.0 - Production
With the Objects option
PL/SQL Release 9.0.1.0.0 - Production
```

Enumerating tables to scan...

```
. process 1 scanning SCOTT.BONUS
. process 1 scanning SCOTT.DEPT
. process 1 scanning SCOTT.EMP
```

Creating Database Scan Summary Report...

Creating Individual Exception Report...

Scanner terminated successfully.

Sample Session of Single Table Scan

The following example shows how to scan a single table to see the effects on migrating it to WE8MSWIN1252. This example assumes the current database character set is in US7ASCII.

Parameter File Method

```
% csscan system/manager parfile=param.txt
```

The `param.txt` file contains the following information:

```
user=scott
table=emp
tochar=we8mswin1252
array=40960
process=1
supress=100
```

Command Line Method

```
% csscan system/manager user=scott table=emp tochar=we8mswin1252 array=40960
process=1 supress=100
```

Scanner Messages

Database Scanner: Release 9.0.1.0.0 - Production

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

Connected to:

Oracle9 Enterprise Edition Release 9.0.1.0.0 - Production

With the Objects option

PL/SQL Release 9.0.1.0.0 - Production

. process 1 scanning SCOTT.EMP

Creating Database Scan Summary Report...

Creating Individual Exception Report...

Scanner terminated successfully.

Scanner Reports

The Scanner generates two reports per scan:

- [Database Scan Summary Report](#)
- [Individual Exception Report](#)

Database Scan Summary Report

A Database Scan Summary Report consists of the following sections. The information available for each section depends on the type of scans and the parameters you select.

- [Database Scanner Parameters](#)
- [Database Size](#)
- [Scan Summary](#)
- [Data Dictionary Conversion Summary](#)
- [Application Data Conversion Summary](#)
- [Application Data Conversion Summary per Column Size Boundary](#)
- [Distribution of Convertible Data per Table](#)
- [Distribution of Convertible Data per Column](#)
- [Indexes To Be Rebuilt](#)

Database Scanner Parameters

This section describes the parameters selected and the type of scan you chose. The following is an example:

Parameter	Value
Scan type	Full database
Scan CHAR data?	YES
Current database character set	WE8ISO8859P1
New database character set	UTF8
Scan NCHAR data?	NO
Array fetch buffer size	102400
Number of processes	4

Database Size

This section describes the current database size. The following is an example:

TABLESPACE	Total(MB)	Used(MB)	Free(MB)
APPS_DATA	1,340.000	1,331.070	8.926
CTX_DATA	30.000	3.145	26.852
INDEX_DATA	140.000	132.559	7.438

RBS_DATA	310.000	300.434	9.563
SYSTEM_DATA	150.000	144.969	5.027
TEMP_DATA	160.000		159.996
TOOLS_DATA	35.000	22.148	12.848
USERS_DATA	220.000	142.195	77.801

Total	2,385.000	2,073.742	311.227

Scan Summary

This indicates the feasibility of the database character set migration. There are two basic criteria that determine the feasibility of the character set migration of the database. One is the condition of the data dictionary and the other is the condition of the application data.

The Scan Summary section consists of two status lines. Depending on the scan mode and the result returned, the following statuses are printed:

- For the data dictionary:
 - All character-type data in the data dictionary remains the same in the new character set
 - All character-type data in the data dictionary is convertible to the new character set
 - Some character-type data in the data dictionary is not convertible to the new character set
- For application data:
 - All character-type application data remains the same in the new character set
 - All character-type application data is convertible to the new character set
 - Some character-type application data is not convertible to the new character set

When all data remains the same in the new character set, it means that the data encoding of the original character set is identical to the target character set. In this case, the character set can be migrated using the `ALTER DATABASE CHARACTER SET` statement.

If all the data is convertible to the new character set, it means that the data can be safely migrated using the Export and Import utilities. However, the migrated data may or may not have the same encoding as the original character set.

See Also: ["Individual Exception Report"](#) on page 10-33 for more information about non-convertible data

The following is sample output:

All character type data in the data dictionary remains the same in the new character set
All character type application data remains the same in the new character set

Data Dictionary Conversion Summary

This section contains the statistics on the conversion summary of the data dictionary. The granularity of this report is per datatype. The following statuses are available:

Table 10–1 Data Conversion Summary for Data Dictionary

Status	Description
Changeless	Number of data cells that remain the same in the new character set
Convertible	Number of data cells that will be successfully converted to the new character set
Exceptional	Number of data cells that cannot be converted. If you choose to convert anyway, some characters will be lost or data will be truncated

If the numbers in both the `Convertible` and `Exceptional` columns are zero, it means that all the data in the data dictionary will remain the same in the new character set.

If the numbers in the `Exceptional` column are zero and some numbers in the `Convertible` columns are non-zero, it means all data in the data dictionary is convertible to the new character set. During import, the relevant data will be converted.

If the numbers in the `Exceptional` column are non-zero, it means there is data in the data dictionary that is not convertible. Therefore, it is not feasible to migrate the current database to the new character because the export and import process cannot convert the data into the new character set. For example, you might have a table name with invalid characters or a PL/SQL procedure where a comment line includes data that can not be mapped to the new character set. These changes to schema objects must be corrected manually prior to migration to a new character set.

This information is available only when a full database scan is performed. The following is an example:

Datatype	Changeless	Convertible	Exceptional	Total
-----	-----	-----	-----	-----
VARCHAR2	971,300	1	0	971,301
CHAR	7	0	0	7
LONG	60,325	0	0	60,325
CLOB				
-----	-----	-----	-----	-----
Total	1,031,632	1	0	1,031,633

Application Data Conversion Summary

This section contains the statistics on conversion summary of the application data. The granularity of this report is per datatype. The following statuses are available:

Table 10–2 Data Conversion Summary for Application Data

Status	Description
Changeless	Number of data cells that remain the same in the new character set
Convertible	Number of data cells that will be successfully converted to the new character set
Exceptional	Number of data cells that cannot be converted. If you choose to convert anyway, some characters will be lost or data will be truncated

The following is sample output:

Datatype	Changeless	Convertible	Exceptional	Total
-----	-----	-----	-----	-----
VARCHAR2	23,213,745	1,324	0	23,215,069
CHAR	423,430	0	0	423,430
LONG	8,624	33	0	8,657
CLOB	58,839	11,114	28	69,981
-----	-----	-----	-----	-----
Total	23,704,638	12,471	28	23,717,137

Application Data Conversion Summary per Column Size Boundary

This section contains the conversion summary of the CHAR and VARCHAR2 application data. The granularity of this report is per column size boundaries

specified by the `BOUNDARIES` parameter. The following status is available for each datatype and each boundary:

The granularity of this report is per datatype. The following statuses are available:

Table 10–3 Data Conversion Summary for Columns in Application Data

Status	Description
Changeless	Number of data cells that remain the same in the new character set
Convertible	Number of data cells that will be successfully converted to the new character set
Exceptional	Number of data cells that cannot be converted. If you choose to convert, some characters will be lost or data will be truncated

This information is available only when the `BOUNDARIES` parameter is specified.

The following is sample output:

Datatype	Changeless	Convertible	Exceptional	Total
-----	-----	-----	-----	-----
VARCHAR2(1..10)	1,474,825	0	0	1,474,825
VARCHAR2(11..100)	9,691,520	71	0	9,691,591
VARCHAR2(101..4000)	12,047,400	1,253	0	12,048,653
-----	-----	-----	-----	-----
CHAR(1..10)	423,413	0	0	423,413
CHAR(11..100)	17	0	0	17
CHAR(101..4000)				
-----	-----	-----	-----	-----
Total	23,637,175	1,324	0	23,638,499

Distribution of Convertible Data per Table

This example show how `Convertible` and `Exceptional` data is distributed within the database. The granularity of this report is per table. If the list contains only a few rows, it means the `Convertible` data is localized. If the list contains many rows, it means the `Convertible` data is spread out in the database.

The following is sample output:

USER.TABLE	Convertible	Exceptional
-----	-----	-----
SMG.SOURCE	1	0
SMG.HELP	12	0
SMG.CLOSE_LIST	16	0

SMG.ATTENDEES	8	0
SGT.DR_010_I1T1	7	0
SGT.DR_011_I1T1	7	0
SGT.MRK_SRV_PROFILE	2	0
SGT.MRK_SRV_PROFILE_TEMP	2	0
SGT.MRK_SRV_QUESTION	3	0

Distribution of Convertible Data per Column

This example shows how Convertible and Exceptional data is distributed within the database. The granularity of this report is per column. The following is an example:

USER.TABLE COLUMN	Convertible	Exceptional
SMG.SOURCE SOURCE	1	0
SMG.HELP INFO	12	0
SMG.CLOSE_LIST FNAME	1	0
SMG.CLOSE_LIST LNAME	1	0
SMG.CLOSE_LIST COMPANY	1	0
SMG.CLOSE_LIST STREET	8	0
SMG.CLOSE_LIST CITY	4	0
SMG.CLOSE_LIST STATE	1	0
SMG.ATTENDEES ATTENDEE_NAME	1	0
SMG.ATTENDEES ADDRESS1	3	0
SMG.ATTENDEES ADDRESS2	2	0
SMG.ATTENDEES ADDRESS3	2	0
SGT.DR_010_I1T1 WORD_TEXT	7	0
SGT.DR_011_I1T1 WORD_TEXT	7	0
SGT.MRK_SRV_PROFILE FNAME	1	0
SGT.MRK_SRV_PROFILE LNAME	1	0
SGT.MRK_SRV_PROFILE_TEMP FNAME	1	0
SGT.MRK_SRV_PROFILE_TEMP LNAME	1	0
SGT.MRK_SRV_QUESTION ANSWER	3	0

Indexes To Be Rebuilt

This generates a list of all the indexes that are affected by the database character set migration. These can be rebuilt upon the import of the data. The following is an example:

USER.INDEX on USER.TABLE(COLUMN)

CD2000.COMPANY_IX_PID_BID_NNAME on CD2000.COMPANY(CO_NLS_NAME)
CD2000.I_MASHINE_MAINT_CONT on CD2000.MACHINE(MA_MAINT_CONT#)
CD2000.PERSON_NEWS_SABUN_CONT_CONT on
CD2000.PERSON_NEWS_SABUN_CONT(CONT_BID)
CD2000.PENEWSABUN3_PEID_CONT on CD2000.PE_NEWS_SABUN_3(CONT_BID)

```
PMS2000.CALLS_IX_STATUS_SUPPMGR on PMS2000.CALLS(SUPPMGR)
PMS2000.MAILQUEUE_CHK_SUB_TOM on PMS2000.MAIL_QUEUE(TO_MAIL)
PMS2000.MAILQUEUE_CHK_SUB_TOM on PMS2000.MAIL_QUEUE(SUBJECT)
PMS2000.TMP_IX_COMP on PMS2000.TMP_CHK_COMP(COMP_NAME)
-----
```

Individual Exception Report

An Individual Exception Report consists of the following summaries:

- Database Scan Parameters
- Application Data Individual Exceptions

Database Scan Parameters

This section describes the parameters and the type of scan chosen. The following is an example:

Parameter	Value
-----	-----
Scan type	Full database
Scan CHAR data?	YES
Current database character set	we8mswin1252
New database character set	utf8
Scan NCHAR data?	NO
Array fetch buffer size	102400
Number of rows to heap up for insert	10
Number of processes	1
-----	-----

Application Data Individual Exceptions

This report identifies the data that has exceptions so that this data can then be modified if necessary.

There are two types of exceptions:

- Exceed Column Size
The column size should be extended if the maximum column width has been surpassed. If not, data truncation occurs.
- Lossy Conversion
The data must be corrected before migrating to the new character set, or else the invalid characters will be converted to a replacement character. Replacement

characters are usually specified as ? or ¸ or a similar linguistically-related character.

The following is an example of an individual exception report that illustrates some possible problems when changing the database character set from WE8ISO8859P1 to UTF8:

User: SCOTT
Table: PRODUCT
Column: NAME
Type: VARCHAR2(10)
Number of Exceptions: 2
Max Post Conversion Data Size: 11

ROWID	Exception Type	Size	Cell Data(first 30 bytes)
-----	-----	-----	-----
AAAA2fAAFAABJwQAAG	exceed column size	11	Ährenfeldt
AAAA2fAAFAABJwQAAU	lossy conversion		óráclê8™
AAAA2fAAFAABJwQAAU	exceed column size	11	óráclê8™
-----	-----	-----	-----

The values Ährenfeldt and óráclê8™ exceed the column size (10 bytes) because each of the characters Ä, ó, â, and ë occupies one byte in WE8ISO8859P1 but two bytes in UTF8. The value óráclê8™ has lossy conversion to UTF8 because the trademark sign ™ (code 153) is not a valid WE8ISO8859P1 character. It is a WE8MSWIN1252 character, which is a superset of WE8ISO8859P1.

You can view the data that has an exception by issuing a SELECT statement:

```
SELECT name FROM scott.product
WHERE ROWID= 'AAAA2fAAFAABJwQAAU' ;
```

You can modify the data that has the exception by issuing an UPDATE statement:

```
UPDATE scott.emp SET ename = 'Oracle8 TM'
WHERE ROWID= 'AAAA2fAAFAABJwQAAU' ;
```

Storage and Performance Considerations in the Scanner

This section describes storage and performance issues in the Scanner. It contains the following topics:

- [Storage Considerations](#)
- [Performance Considerations](#)

Storage Considerations

This section describes the sizing and the growth of the Scanner's system tables, and explains the approach to maintain them. There are three system tables that can increase rapidly depending on the nature of the data stored in the database.

- [CSM\\$TABLES](#)
- [CSM\\$COLUMNS](#)
- [CSM\\$ERRORS](#)

CSM\$TABLES

The Scanner enumerates all tables that need to be scanned into the table CSM\$TABLES.

You might want to assign a large tablespace to the user CSMIG by amending the CSMINST.SQL script. By default, the SYSTEM tablespace is assigned to the user CSMIG.

You can look up the number of tables (to get an estimate of how large CSM\$TABLES can become) in the database by issuing the following SQL statement:

```
SELECT COUNT(*) FROM DBA_TABLES;
```

CSM\$COLUMNS

The Scanner stores statistical information for each column scanned into the table CSM\$COLUMNS.

You might want to assign a large tablespace to the user CSMIG by amending the CSMINST.SQL script. By default, the SYSTEM tablespace is assigned to CSMIG user.

You can look up the number of character type columns (to get an estimate of how large CSM\$COLUMNS can become) in the database by issuing the following SQL statement:

```
SELECT COUNT(*) FROM DBA_TAB_COLUMNS
```

```
WHERE DATA_TYPE IN ('CHAR', 'VARCHAR2', 'LONG', 'CLOB');
```

CSM\$ERRORS

When exceptions are detected with cell data, the Scanner inserts individual exception information into the table CSM\$ERRORS. This information then appears in the Individual Exception Report and facilitates identifying records to be modified if necessary.

If your database contains a lot of data that is signaled as `Exceptional` or `Convertible` (when the parameter `CAPTURE=Y` is set), the table CSM\$ERRORS can grow too large. You can prevent the CSM\$ERRORS table from growing unnecessarily large by using the `SUPPRESS` parameter.

The `SUPPRESS` parameter applies to each table. The Scanner suppresses inserting individual `Exceptional` information after the specified number of exceptions is inserted. Limiting the number of exceptions to be recorded may not be useful if the exceptions are spread over different tables.

You might want to assign a large tablespace to the user CSMIG by amending the CSMINST.SQL script.

Performance Considerations

This section describes ways to increase performance when scanning the database.

Utilizing Multiple Scan Processes

If you plan to scan a relatively large database, for example, over 50GB, you might want to consider using multiple scan processes. This shortens the duration time of database scans by utilizing hardware resources such as CPU and memory available on the machine.

Array Fetch Buffer Size

The Scanner fetches multiple rows at a time when an array fetch is allowed. Generally, you will improve performance by letting the Scanner use a bigger array fetch buffer.

Suppressing Exception and Convertible Log

The Scanner inserts individual `Exceptional` and `Convertible` (when `CAPTURE=Y`) information into the table CSM\$ERRORS. In general, insertion into the CSM\$ERRORS table is more costly than data fetching. If your database has a lot of data that is signaled as `Exceptional` or `Convertible`, the Scanner issues many

insert statements, causing performance degradation. Oracle Corporation recommends setting a limit on the number of exception rows to be recorded using the SUPRESS parameter.

Scanner Utility Reference Material

This section contains the following reference material:

- [Scanner Views](#)
- [Scanner Messages](#)

Scanner Views

The Scanner uses the following views.

CSMV\$COLUMNS

This view contains statistical information of columns that were scanned.

Column	Datatype	NULL	Description
OWNER_ID	NUMBER	NOT NULL	Userid of the table owner
OWNER_NAME	VARCHAR2 (30)	NOT NULL	User name of the table owner
TABLE_ID	NUMBER	NOT NULL	Object ID of the table
TABLE_NAME	VARCHAR2 (30)	NOT NULL	Object name of the table
COLUMN_ID	NUMBER	NOT NULL	Column ID
COLUMN_INTID	NUMBER	NOT NULL	Internal column id (for ADT)
COLUMN_NAME	VARCHAR2 (30)	NOT NULL	Column name
COLUMN_TYPE	VARCHAR2 (9)	NOT NULL	Column Datatype
TOTAL_ROWS	NUMBER	NOT NULL	Number of rows in this table
NULL_ROWS	NUMBER	NOT NULL	Number of NULL data cells
CONV_ROWS	NUMBER	NOT NULL	Number of data cells that need to be converted
ERROR_ROWS	NUMBER	NOT NULL	Number of data cells that have exceptions
EXCEED_SIZE_ROWS	NUMBER	NOT NULL	Number of data cells that have exceptions
DATA_LOSS_ROWS	NUMBER		Number of data cells that undergo lossy conversion
MAX_POST_CONVERT_SIZE	NUMBER		Maximum post-conversion data size

CSMV\$CONSTRAINTS

This view contains statistical information of columns that were scanned.

Column	Datatype	NULL	Description
OWNER_ID	NUMBER	NOT NULL	Userid of the constraint owner
OWNER_NAME	VARCHAR2(30)	NOT NULL	User name of the constraint owner
CONSTRAINT_ID	NUMBER	NOT NULL	Object ID of the constraint
CONSTRAINT_NAME	VARCHAR2(30)	NOT NULL	Object name of the constraint
CONSTRAINT_TYPE#	NUMBER	NOT NULL	Constraint type number
CONSTRAINT_TYPE	VARCHAR2(11)	NOT NULL	Constraint type name
TABLE_ID	NUMBER	NOT NULL	Object ID of the table
TABLE_NAME	VARCHAR2(30)	NOT NULL	Object name of the table
CONSTRAINT_RID	NUMBER	NOT NULL	Root constraint id
CONSTRAINT_LEVEL	NUMBER	NOT NULL	Constraint level

CSMV\$ERRORS

This view contains individual exception information of cell data and object definitions.

Column	Datatype	NULL	Description
OWNER_ID	NUMBER	NOT NULL	Userid of the table owner
OWNER_NAME	VARCHAR2(30)	NOT NULL	User name of the table owner
TABLE_ID	NUMBER	NOT NULL	Object ID of the table
TABLE_NAME	VARCHAR2(30)		Object name of the table
COLUMN_ID	NUMBER		Column ID
COLUMN_INTID	NUMBER		Internal column ID (for ADT)
COLUMN_NAME	VARCHAR2(30)		Column name
DATA_ROWID	VARCHAR2(1000)		The rowid of the data
COLUMN_TYPE	VARCHAR2(9)		Column datatype of object type
ERROR_TYPE	VARCHAR2(11)		Type of error encountered

CSMV\$INDEXES

This view contains individual exception information of cell data and object definitions.

Column	Datatype	NULL	Description
INDEX_OWNER_ID	NUMBER	NOT NULL	Userid of the index owner
INDEX_OWNER_NAME	VARCHAR2 (30)	NOT NULL	User name of the index owner
INDEX_ID	NUMBER	NOT NULL	Object ID of the index
INDEX_NAME	VARCHAR2 (30)		Object name of the index
INDEX_STATUS#	NUMBER		Status number of the index
INDEX_STATUS	VARCHAR2 (8)		Status of the index
TABLE_OWNER_ID	NUMBER		Userid of the table owner
TABLE_OWNER_NAME	VARCHAR2 (30)		User name of the table owner
TABLE_ID	NUMBER		Object ID of the table
TABLE_NAME	VARCHAR2 (30)		Object name of the table
COLUMN_ID	NUMBER		Column ID
COLUMN_INTID	NUMBER		Internal column ID (for ADT)
COLUMN_NAME	VARCHAR2 (30)		Column name

CSMV\$TABLES

This view contains information about database tables to be scanned. The Scanner enumerates all tables to be scanned into this view.

Column	Datatype	NULL	Description
OWNER_ID	NUMBER	NOT NULL	Userid of the table owner
OWNER_NAME	VARCHAR2 (30)	NOT NULL	User name of the table owner
TABLE_ID	NUMBER		Object ID of the table
TABLE_NAME	VARCHAR2 (30)		Object name of the table
MIN_ROWID	VARCHAR2 (18)		Minimum rowid of the split range of the table
MAX_ROWID	VARCHAR2 (18)		Maximum rowid of the split range of the table
BLOCKS	NUMBER		Number of blocks in the split range
SCAN_COLUMNS	NUMBER		Number of columns to be scanned
SCAN_ROWS	NUMBER		Number of rows to be scanned
SCAN_START	VARCHAR2 (8)		Time table scan started
SCAN_END	VARCHAR2 (8)		Time table scan completed

Scanner Messages

The Scanner has the following error messages:

CSS-00100 failed to allocate memory size of number
An attempt was made to allocate memory with size 0 or bigger than the maximum size.
This is an internal error. Contact Oracle Customer Support.

CSS-00101 failed to release memory
An attempt was made to release memory with invalid pointer.
This is an internal error. Contact Oracle Customer Support.

CSS-00102 failed to release memory, null pointer given
An attempt was made to release memory with null pointer.
This is an internal error. Contact Oracle Customer Support.

CSS-00105 failed to parse BOUNDARIES parameter
BOUNDARIES parameter was specified in an invalid format.
Refer to the manual for the correct syntax.

CSS-00106 failed to parse SPLIT parameter
SPLIT parameter was specified in an invalid format.
Refer to the manual for the correct syntax.

CSS-00107 Character set migration utility schem not installed
CSM\$VERSION table not found in the database.
Run CSMINST.SQL on the database.

CSS-00108 Character set migration utility schema not compatible
Incompatible CSM\$* tables found in the database.
Run CSMINST.SQL on the database.

CSS-00110 failed to parse userid
USERID parameter was specified in an invalid format.
Refer to the manual for the correct syntax.

CSS-00111 failed to get RDEMS version
Failed to retrieve the value of the Version of the database.
This is an internal error. Contact Oracle Customer Support.

CSS-00112 database version not supported
The database version is older than release 8.0.5.0.0.
Upgrade the database to release 8.0.5.0.0 or later, then try again.

CSS-00113 user %s is not allowed to access data dictionary
The specified user cannot access the data dictionary.
Set O7_DICTIONARY_ACCESSIBILITY parameter to TRUE, or use SYS user.

CSS-00114 failed to get database character set name
Failed to retrieve value of NLS_CHARACTERSET or NLS_NCHAR_CHARACTERSET parameter from NLS_

DATABASE_PARAMETERS view.

This is an internal error. Contact Oracle Customer Support.

CSS-00115 invalid character set name %s

The specified character set is not a valid Oracle character set.

Refer to [Appendix A, "Locale Data"](#) for the correct character set name.

CSS-00116 failed to reset NLS_LANG/NLS_NCHAR parameter

Failed to force NLS_LANG character set to be same as database character set.

This is an internal error. Contact Oracle Customer Support.

CSS-00117 failed to clear previous scan log

Failed to delete all rows from CSM\$* tables.

This is an internal error. Contact Oracle Customer Support.

CSS-00118 failed to save command parameters

Failed to insert rows into CSM\$PARAMETERS table.

This is an internal error. Contact Oracle Customer Support.

CSS-00119 failed to save scan start time

Failed to insert a row into CSM\$PARAMETERS table.

This is an internal error. Contact Oracle Customer Support.

CSS-00120 failed to enumerate tables to scan

Failed to enumerate tables to scan into CSM\$TABLES table.

This is an internal error. Contact Oracle Customer Support.

CSS-00121 failed to save scan complete time

Failed to insert a row into CSM\$PARAMETERS table.

This is an internal error. Contact Oracle Customer Support.

CSS-00122 failed to create scan report

Failed to create database scan report.

This is an internal error. Contact Oracle Customer Support.

CSS-00123 failed to check if user %s exist

Select statement that checks if the specified user exists in the database failed.

This is an internal error. Contact Oracle Customer Support.

CSS-00124 user %s not found

The specified user does not exist in the database.

Check the user name.

CSS-00125 failed to check if table %s.%s exist

Select statement that checks if the specified table exists in the database failed.

This is an internal error. Contact Oracle Customer Support.

CSS-00126 table %s.%s not found

The specified table does not exist in the database.

Check the user name and table name.

CSS-00127 user %s does not have DBA privilege
The specified user does not have DBA privileges, which are required to scan the database.
Choose a user with DBA privileges.

CSS-00128 failed to get server version string
Failed to retrieve the version string of the database.
None.

CSS-00130 failed to initialize semaphore
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00131 failed to spawn scan process %d
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00132 failed to destroy semaphore
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00133 failed to wait semaphore
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00134 failed to post semaphore
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00140 failed to scan table (tid=%d, oid=%d)
Data scan on this particular table failed.
This is an internal error. Contact Oracle Customer Support.

CSS-00141 failed to save table scan start time
Failed to update a row in the CSM\$TABLES table.
This is an internal error. Contact Oracle Customer Support.

CSS-00142 failed to get table information
Failed to retrieve various information from user id and object id of the table.
This is an internal error. Contact Oracle Customer Support.

CSS-00143 failed to get column attributes
Failed to retrieve column attributes of the table.
This is an internal error. Contact Oracle Customer Support.

CSS-00144 failed to scan table %s.%s
Data scan on this particular table was not successful.
This is an internal error. Contact Oracle Customer Support.

CSS-00145 failed to save scan result for columns
Failed to insert rows into CSM\$COLUMNS table.
This is an internal error. Contact Oracle Customer Support.

CSS-00146 failed to save scan result for table
Failed to update a row of CSM\$TABLES table.
This is an internal error. Contact Oracle Customer Support.

CSS-00147 unexpected data truncation
Scanner allocates the exactly same size of memory as the column byte size for fetch buffer, resulting in unexpected data truncation.
This is an internal error. Contact Oracle Customer Support.

CSS-00150 failed to enumerate table
Failed to retrieve the specified table information.
This is an internal error. Contact Oracle Customer Support.

CSS-00151 failed to enumerate user tables
Failed to enumerate all tables that belong to the specified user.
This is an internal error. Contact Oracle Customer Support.

CSS-00152 failed to enumerate all tables
Failed to enumerate all tables in the database.
This is an internal error. Contact Oracle Customer Support.

CSS-00153 failed to enumerate character type columns
Failed to enumerate all CHAR, VARCHAR2, LONG, and CLOB columns of tables to scan.
This is an internal error. Contact Oracle Customer Support.

CSS-00154 failed to create list of tables to scan
Failed to enumerate the tables into CSM\$TABLES table.
This is an internal error. Contact Oracle Customer Support.

CSS-00155 failed to split tables for scan
Failed to split the specified tables.
This is an internal error. Contact Oracle Customer Support.

CSS-00156 failed to get total number of tables to scan
Select statement that retrieves the number of tables to scan failed.
This is an internal error. Contact Oracle Customer Support.

CSS-00157 failed to retrieve list of tables to scan
Failed to read all table ids into the scanner memory.
This is an internal error. Contact Oracle Customer Support.

CSS-00158 failed to retrieve index defined on column
Select statement that retrieves index defined on the column fails.
This is an internal error. Contact Oracle Customer Support.

CSS-00160 failed to open summary report file
File open function returned error.
Check if you have create/write privilege on the disk and check if the file name specified for the LOG parameter is valid.

CSS-00161 failed to report scan elapsed time
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00162 failed to report database size information
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00163 failed to report scan parameters
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00164 failed to report Scan summary
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00165 failed to report conversion summary
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00166 failed to report convertible data distribution
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00167 failed to open exception report file
File open function returned error.
Check if you have create/write privilege on the disk and check if the file name specified for LOG parameter is valid.

CSS-00168 failed to report individual exceptions
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00170 failed to retrieve size of tablespace %
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00171 failed to retrieve free size of tablespace %s
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00172 failed to retrieve total size of tablespace %s
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00173 failed to retrieve used size of the database
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00174 failed to retrieve free size of the database
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00175 failed to retrieve total size of the database
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00176 failed to enumerate user tables in bitmapped tablespace
Failed to enumerate tables in bitmapped tablespace.
This is an internal error. Contact Oracle Customer Support.

Oracle Locale Builder Utility

This chapter describes the Oracle Locale Builder Utility. It includes the following topics:

- [Overview of the Locale Builder Utility](#)
- [Setting the Language Definition with the Locale Builder](#)
- [Setting the Territory Definition with the Locale Builder](#)
- [Setting the Character Set Definition with the Locale Builder](#)
- [Sorting with the Locale Builder](#)

Overview of the Locale Builder Utility

The Locale Builder offers an easy and efficient way to access and define NLS locale data definitions. It provides a graphical user interface through which you can easily view, modify, and define locale-specific data. It extracts data from the text and binary definition files and presents them in a readable format, so you can process the information without worrying about the specific definition formats used in these files.

The Locale Builder handles four types of locale definitions: language, territory, character set, and linguistic sort. It also supports user-defined characters and customized linguistic rules. You can view definitions in existing text and binary definition files and make changes to them or create your own definitions.

Configuring Unicode Fonts for the Locale Builder

The Locale builder uses Unicode characters in many of its functions. For example, it shows the mapping of local character codepoints to Unicode codepoints. Therefore, Oracle Corporation recommends that you use a Unicode font to fully support the Locale Builder. If a character cannot be rendered with your local fonts, it will probably be displayed as an empty box.

Font Configuration on Windows

There are many Windows TrueType and OpenType fonts that support Unicode. Oracle Corporation recommends using the Arial Unicode MS from Microsoft, because it includes about 51,000 glyphs and covers most of the characters in Unicode 3.0.

After installing the Unicode font, add the font to the Java Runtime so it can be used by the Oracle Locale Builder. The Java Runtime uses a font configuration file to map predefined Java virtual fonts to fonts that are available on Windows. The name of the configuration file is `font.properties` and it is located in the `$JAVAHOME/lib` directory. For example, to include the installed Arial Unicode MS font, add the following entry to the `font.properties` file:

```
dialog.n = Arial Unicode MS, DEFAULT_CHARSET
```

where *n* is next available sequence number to which you want to assign the Arial Unicode MS font in the font list. Java Runtime looks through the font mapping list for each virtual font and use the first font available on your system.

Add an entry for the new font to each font mapping list that you want the new font to be used for. After editing the `font.properties` file, restart the Locale Builder so it can use the new fonts.

Note: For a detailed description of the `font.properties` file format, visit Sun's internationalization website.

Font Configuration on Other Platforms

In general, there are fewer choices of Unicode fonts for non-Windows platforms than for Windows platforms. If you cannot find a Unicode font with satisfactory character coverage, you can use multiple fonts to cover the different languages. For each font that you want to add to the Java Runtime, install the font and add the font entries into the `font.properties` file using the steps described above for the Windows platform.

For example, to display Japanese characters on Sun Solaris using the font `ricoh-hg mincho`, add an entry to the existing `font.properties` file in `$JAVAHOME/lib`.

```
serif.plain.0=-monotype-times new roman-regular-r---%d-*-p-*-iso8859-1
serif.plain.1=-urw-itc
zapfdingbats-medium-r-normal--%d-*-p-*-sun-fontspecific
serif.plain.2=-*-symbol-medium-r-normal--%d-*-p-*-sun-fontspecific
serif.plain.3=-ricoh-hg mincho l-medium-r-normal--%d-*-m-*-jisx0201.1976-0
```

For font availability, refer to your operating system specific documentation.

The Locale Builder Interface

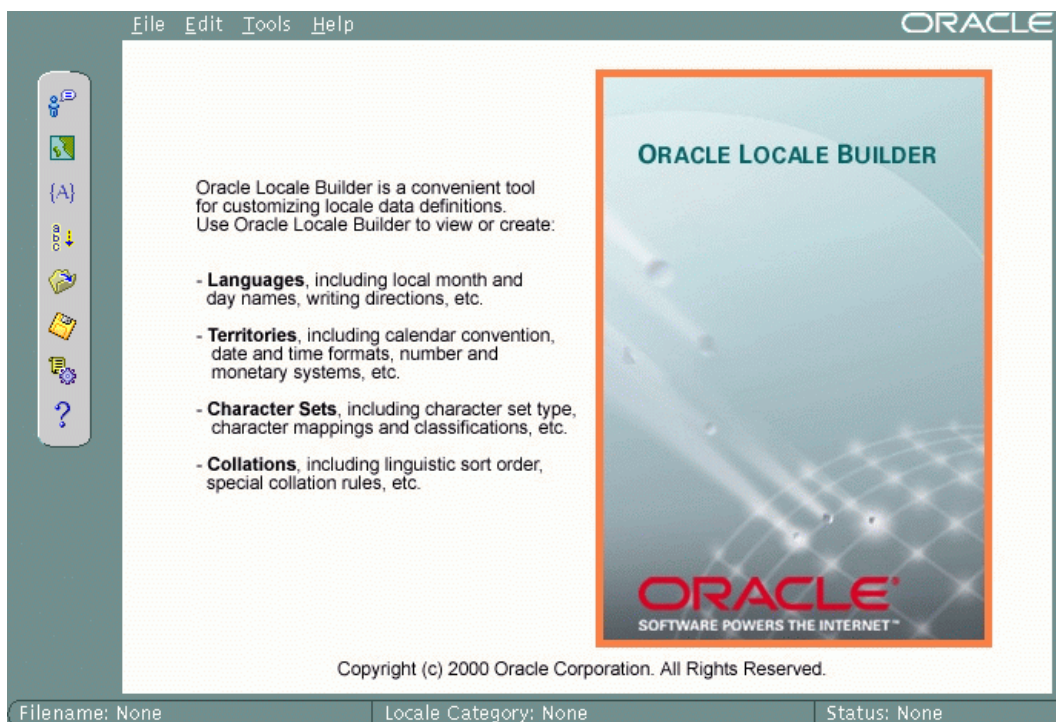
Ensure that the `ORACLE_HOME` initialization parameter is set before starting the Builder.

Start the Locale Builder at the Unix prompt by issuing the following command:

```
% lbuilder
```

After you start the Locale Builder, the screen illustrated in [Figure 11-1](#) appears.

Figure 11–1 *Locale Builder Utility*



Locale Builder General Screens

Before beginning with specific tasks, you might want to become familiar with the general screens that you can use at different times. These screens are:

- [Existing Definitions Dialog Box](#)
under the General tab
- [Session Log Dialog Box](#)
under the Tools menu
- [Previewing the NLT File Dialog Box](#)
as a tab in many tasks
- [Open File Dialog Box](#)
under the File menu

Note: Oracle Locale Builder includes online help.

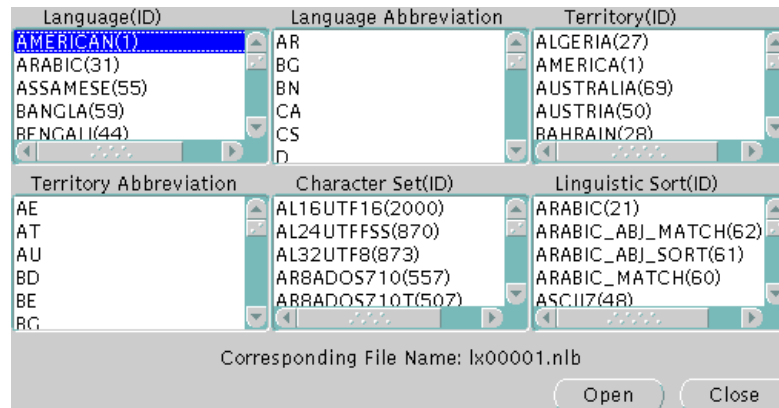
Restrictions

The following restrictions apply when choosing locale object names:

- Names must be all ASCII characters
- Names must start with a letter
- Language, territory, and character set names cannot contain underscores

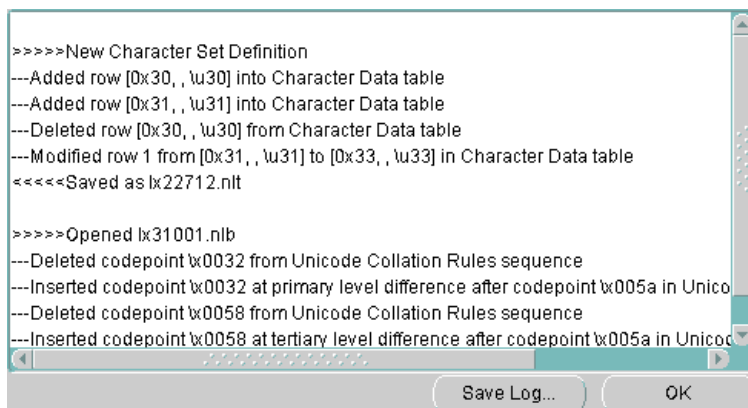
Note: Only certain ID ranges are valid values for the user-defined LANGUAGE, TERRITORY, CHARACTER SET, MONOLINGUAL COLLATION, and MULTILINGUAL COLLATION definitions. They are listed in the text about relevant screenshots.

Figure 11–2 Existing Definitions Dialog Box



The Existing Definitions dialog box allows you to open locale objects by name. If you know a specific language, territory, linguistic sort (collation), or character set that you want to start with, click on the displayed value. For example, you can open the AMERICAN language definition file, as shown in [Figure 11–2](#). In this case, you will open the lx00001.nlb file.

Abbreviations are for reference only and cannot be opened.

Figure 11–3 Session Log Dialog Box

The Session Log dialog box shows what actions have been taken in a given session. This way, you can keep a record of all changes and, if necessary, undo or modify past changes. [Figure 11–3](#) illustrates a typical example.

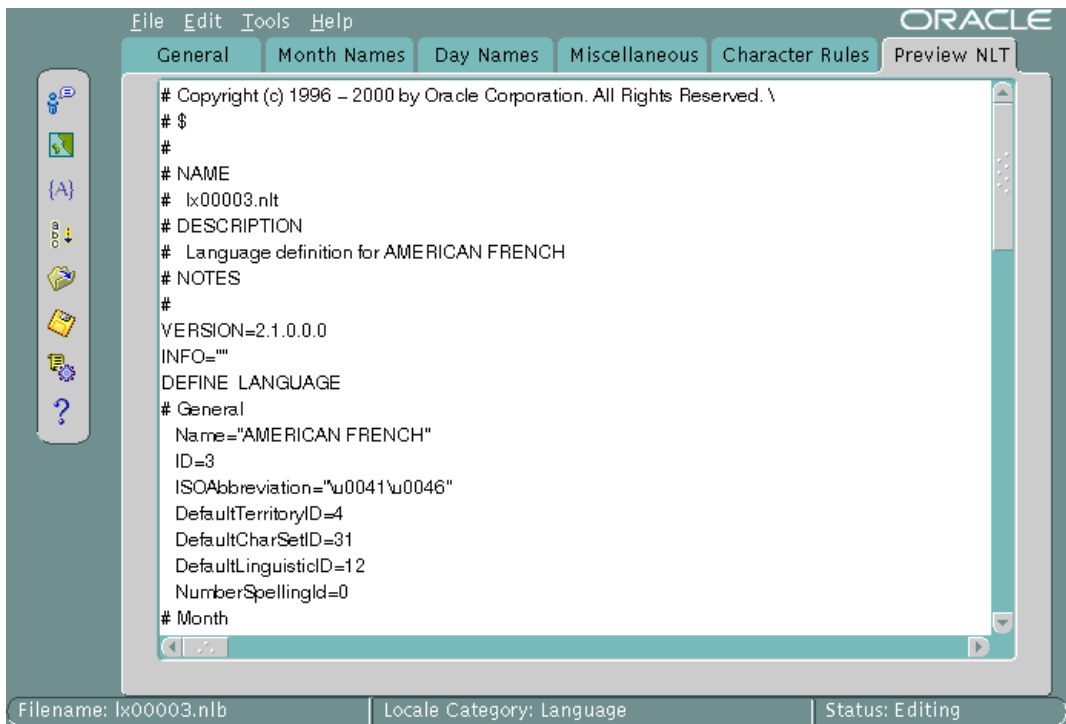
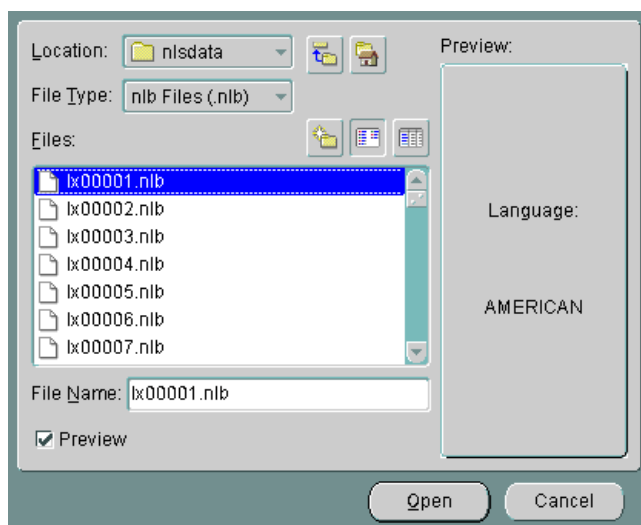
Figure 11–4 *Previewing the NLT File Dialog Box*

Figure 11–4 illustrates viewing an NLT file. It is a text file with the file extension `.nlt` which shows the settings for a specific language, territory, character set, or linguistic sort are kept. The NLT file is not modifiable from this dialog box. Instead, the purpose is to present an easily readable form of the file for you to see if your changes look correct. You must use the specific elements of the Locale Builder to modify the NLT file.

Figure 11–5 Open File Dialog Box



The Open File dialog box opens an NLB file so you can modify it or use it as a template. The NLB file is a binary file with the file extension `.nlb` that contains the binary equivalent of the information in the NLT file. [Figure 11–5](#) illustrates opening `lx00001.nlb`, which is for the language definition for `AMERICAN`. By highlighting Preview, you can see what type of NLB file you have selected.

Setting the Language Definition with the Locale Builder

This section will use a sample scenario of creating a new language based on French. This new language will be called `AMERICAN FRENCH`. First, you need to open `FRENCH` from the Existing Definitions dialog box. [Figure 11–6](#) illustrates the first screen.

Figure 11–6 Language General Information

The screenshot shows the Oracle Locale Builder utility window. The 'General' tab is selected, displaying the following fields and values:

Field	Value
Language Name:	AMERICAN FRENCH
Language ID:	1001
Language Abbreviation:	AF
Default definitions for this language:	
Default Territory:	FRANCE
Default Character Set:	WE8ISO8859P1
Default Linguistic Definition:	FRENCH

Below the fields is a button labeled 'Show Existing Definitions...'. The status bar at the bottom indicates 'Filename: lx00003.nlb', 'Locale Category: Language', and 'Status: Editing'.

Figure 11–6 illustrates a user-defined setting of AMERICAN FRENCH and a user-defined abbreviation of AF. The ISO Abbreviation field is not limited to standard ISO abbreviations, so you can create your own: AF, in this case. The Default settings are inherited and optional. You can build upon an inherited setting and modify it to add additional properties.

The valid range for the language ID field for a user-defined language is 1,000 to 10,000.

Figure 11–7 Language Definition Month Information

Capitalize initial letter of month names?

☒ Yes ☐ No (or non-applicable)

	Full Month Names	Abbreviated Month Names
Month 01:	january	jan
Month 02:	février	fev
Month 03:	mars	mar
Month 04:	avril	avr
Month 05:	mai	mai
Month 06:	juin	jun
Month 07:	juillet	jul
Month 08:	août	aou
Month 09:	septembre	sep
Month 10:	octobre	oct
Month 11:	novembre	nov
Month 12:	décembre	dec

Filename: lx00003.nlb Locale Category: Language Status: Editing

Figure 11–7 illustrates how to set month names using the Month Names tab. All names are shown as they appear in the NLT file. If you set `NLS_LANG` to `AMERICAN FRENCH`, the rules shown in the figure apply.

Figure 11–8 Language Definition Type Information

	Full Day Names	Abbreviated Day Names
Sunday:	sunday	sun
Monday:	lundi	lu
Tuesday:	mardi	ma
Wednesday:	mercredi	me
Thursday:	jeudi	je
Friday:	vendredi	ve
Saturday:	samedi	sa

Filename: lx00003.nlb Locale Category: Language Status: Editing

Figure 11–8 illustrates the Day Names tab, which allows you to choose default day names. All names are shown as they appear in the NLT file. If you set `NLS_LANG` to `AMERICAN FRENCH`, the rules in the figure apply.

Setting the Territory Definition with the Locale Builder

This section will use a sample scenario of creating a new territory called `REDWOOD SHORES`, and use `RS` as an abbreviation for it. In this case, we will create a new definition that is not based on an existing one.

The basic tasks are to assign a name and choose calendar, number, date/time, and currency formats. Figure 11–9 illustrates how to begin.

Figure 11–9 Territory Definition General Information

The screenshot shows the Oracle Locale Builder interface. At the top, there is a menu bar with 'File', 'Edit', 'Tools', and 'Help'. Below the menu bar is a tabbed interface with tabs for 'General', 'Calendar', 'Date&Time', 'Number', 'Monetary', and 'Preview NLT'. The 'General' tab is currently selected. On the left side of the 'General' tab, there is a vertical toolbar with icons for creating a new territory, opening an existing one, saving, and other functions. The main area of the 'General' tab contains three text input fields: 'Territory Name' with the value 'REDWOOD SHORES', 'Territory ID' with the value '1001', and 'Territory Abbreviation' with the value 'RS'. Below these fields is a button labeled 'Show Existing Definitions...'. At the bottom of the dialog, there is a status bar with three sections: 'Filename: Untitled', 'Locale Category: Territory', and 'Status: Editing'.

Field	Value
Territory Name	REDWOOD SHORES
Territory ID	1001
Territory Abbreviation	RS

In [Figure 11–9](#), we have manually inserted REDWOOD SHORES and RS for a new territory.

The valid range for the territory ID field for a user-defined territory is 1,000 to 10,000.

See Also: [Chapter 3, "Setting Up a Globalization Support Environment"](#)

Figure 11–10 Territory Definition Calendar

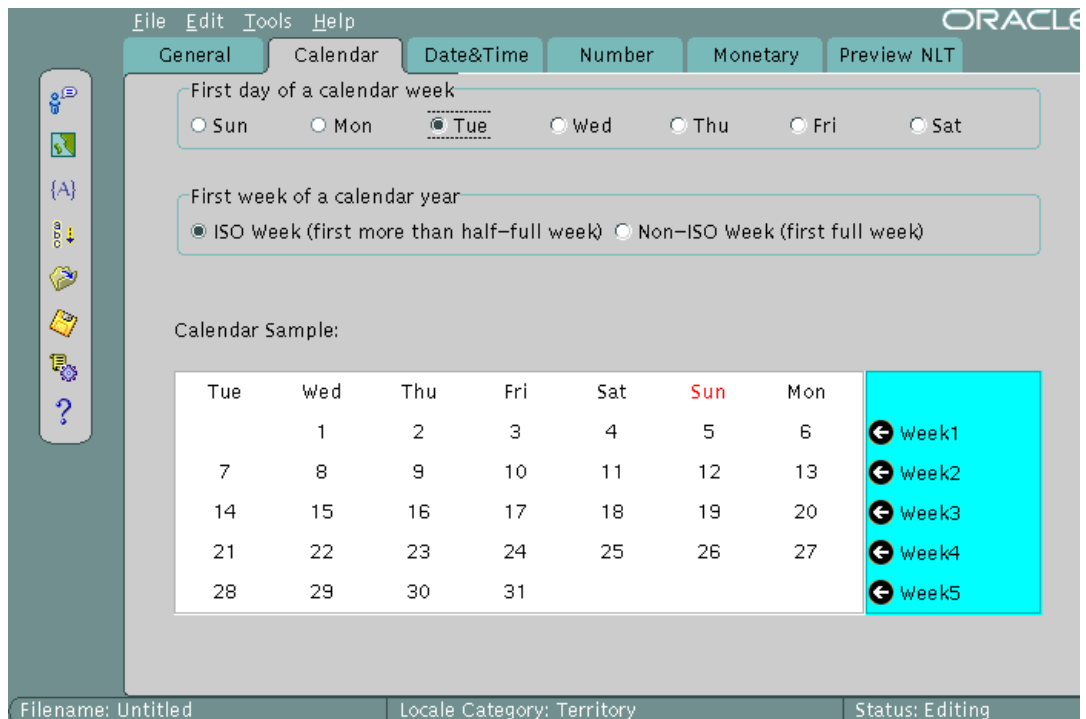


Figure 11–10 illustrates how to set Calendar characteristics. Clicking on a radio button causes the Calendar Sample to display sample output. In this case, Tuesday is the first day of the week.

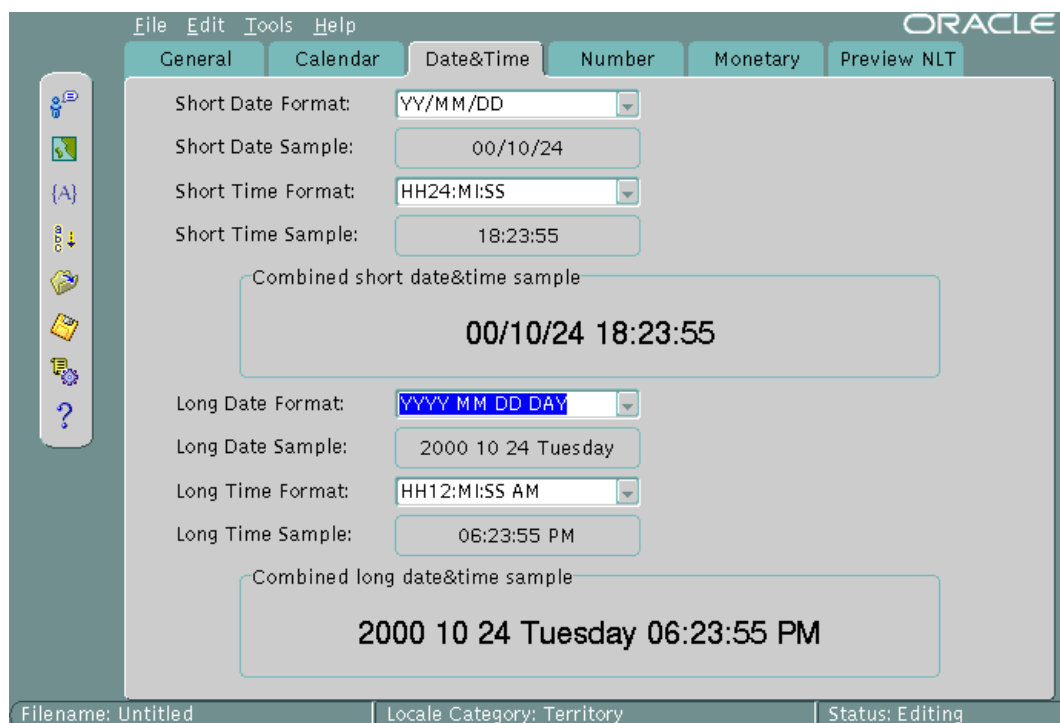
Figure 11–11 Territory Definition Date and Time Conventions

Figure 11–11 illustrates typical date and time settings. Sample formats are displayed when you choose a setting from the drop-down menus. In this case, we set the default date format for REDWOOD_SHORES to YY/MM/DD instead of the typical territory default of DD-MM-YY.

You can also create your own formats instead of using the selection from the drop-down menus.

Figure 11–12 Territory Definition Number Conventions

The screenshot shows the Oracle Locale Builder interface with the 'Number' tab selected. The settings are as follows:

- Decimal Symbol:** .
- Negative Sign Location:** -100 (selected)
- Numeric Group Separator:** ,
- Number Grouping:** 4
- Number Sample:** -1,2345.12
- List Separator:** ,
- Measurement System:** Metric
- Rounding Indicator (value greater than which to round up):** 4
- Rounding Sample:** 10.4 is rounded to 10 and 10.5 is rounded to 11

The bottom status bar shows: Filename: Untitled | Locale Category: Territory | Status: Editing

Figure 11–12 illustrates typical number settings. Sample formats are displayed when you choose a setting from the drop-down menus. The default for number grouping is 3, but 4 is used in this case.

You can type your own values instead of using the drop-down menus.

Figure 11–13 Territory Definition Monetary Conventions

The screenshot shows the Oracle Locale Builder interface with the 'Monetary' tab selected. The 'General' tab is also visible. The 'Monetary' tab contains the following settings:

- Local Currency Symbol: \$
- Alternative Currency Symbol: €
- Currency Presentation: -\$100
- Decimal Symbol: .
- Group Separator: ,
- Monetary Number Grouping: 3
- Monetary Precision: 3
- Credit Symbol: +
- Debit Symbol: -

Below these settings, there are two preview boxes:

- Credit: + \$ 1,234.123
- Debit: - \$ 1,234.123

At the bottom of the Monetary tab, there are two more settings:

- International Currency Separator: (blank space)
- International Currency Symbol: USD

A large preview box at the bottom of the Monetary tab displays the formatted number: 1,234 USD.

The bottom status bar shows: Filename: Untitled | Locale Category: Territory | Status: Editing

Figure 11–13 illustrates how to set monetary conventions for territories. Note that the default International Currency Separator is a blank space, so it is not visible in the screen. In this case, we chose the Euro as an alternate currency symbol.

You can type your own values instead of using the drop-down menus.

Setting the Character Set Definition with the Locale Builder

In some cases, you may wish to tailor a character set to meet specific user needs. In Oracle9i, you can extend an existing encoded character set definition to suit your needs. User-defined characters are often used to encode special characters representing:

- Proper names
- Historical Han characters that are not defined in an existing character set standard

- Vendor-specific characters
- New symbols or characters you define

This section describes how Oracle supports user-defined character. It describes:

- [Character Sets with User-Defined Characters](#)
- [Oracle's Character Set Conversion Architecture](#)
- [Unicode 3.1 Private Use Area](#)
- [UDC Cross References](#)

Character Sets with User-Defined Characters

User-defined characters are typically supported within East Asian character sets. These East Asian character sets have at least one range of reserved codepoints for use as user-defined characters. For example, Japanese Shift JIS preserves 1880 codepoints for user-defined characters as follows:

Table 11–1 *Shift JIS Codepoint Example*

Japanese Shift JIS UDC Range	Number of Codepoints
F040-F07E, F080-F0FC	188
F140-F17E, F180-F1FC	188
F240-F27E, F280-F2FC	188
F340-F37E, F380-F3FC	188
F440-F47E, F480-F4FC	188
F540-F57E, F580-F5FC	188
FF640-F67E, F680-F6FC	188
F740-F77E, F780-F7FC	188
F840-F87E, F880-F8FC	188
F940-F97E, F980-F9FC	188

The Oracle character sets listed in [Table 11–2](#) contain pre-defined ranges that allow you to support user-defined characters:

Table 11–2 Oracle Character Sets with UDC

Character Set Name	Number of UDC Codepoints Available
JA16DBCS	4370
JA16EBDIC930	4370
JA16SJIS	1880
JA16SJISYEN	1880
KO16DBCS	1880
KO16MSWIN949	1880
ZHS16DBCS	1880
ZHS16GBK	2149
ZHT16DBCS	6204
ZHT16MSWIN950	6217

Oracle's Character Set Conversion Architecture

The codepoint value that represents a particular character may vary among different character sets. For example, the Japanese kanji character:

Figure 11–14 Kanji Example

亜

is encoded as follows in different Japanese character sets:

Table 11–3 Kanji Example with Character Conversion

Character Set	Unicode	JA16SJIS	JA16EUC	JA16DBCS
Character Value of	4E9C	889F	B0A1	4867
亜				

In Oracle, all character sets are defined in terms of a Unicode 3.0 code point. That is, each character is defined as a Unicode 3.0 code value. Character conversion takes place transparently to users by using Unicode as the intermediate form. For

example, when a JA16SJIS client connects to a JA16EUC database, the character shown in [Figure 11–14, "Kanji Example"](#) (value 889F) entered from the JA16SJIS client is internally converted to Unicode (value 4E9C), and then converted to JA16EUC(value B0A1).

Unicode 3.1 Private Use Area

Unicode 3.0 reserves the range E000-F8FF for the Private Use Area (PUA). The PUA is intended for private use character definition by end users or vendors.

User-defined characters can be converted between two Oracle character sets by using Unicode 3.0 PUA as the intermediate form, the same as standard characters.

UDC Cross References

User-defined character cross references between Japanese character sets, Korean character sets, Simplified Chinese character sets and Traditional Chinese character sets are contained in the following distribution sets:

```
${ORACLE_HOME}/ocommon/nls/demo/udc_ja.txt  
${ORACLE_HOME}/ocommon/nls/demo/udc_ko.txt  
${ORACLE_HOME}/ocommon/nls/demo/udc_zhs.txt  
${ORACLE_HOME}/ocommon/nls/demo/udc_zht.txt
```

These cross references are useful when registering user-defined characters across operating systems. For example, when registering a new user-defined character on both a Japanese Shift-JIS operating system and a Japanese IBM Host operating system, you may want to pick up F040 on Shift-JIS operating system and 6941 on IBM Host operating system for the new user-defined character so that Oracle can convert correctly between JA16SJIS and JA16DBCS. You can find out that both Shift-JIS UDC value F040 and IBM Host UDC value 6941 are mapped to the same Unicode PUA value E000 in the user-defined character cross reference.

See Also: [Appendix B, "Unicode Character Code Assignments"](#)
for more information about customizing a character set definition file

Character Set Definition File Conventions

By default, the Locale Builder generates the next available character set name for you. You can, however, generate your own character set name. You should follow certain conventions when creating a character set. In particular, the convention used

for naming character set definition NLT files is the format: `1x2dddd.nlt`, where `dddd` = 4 digit Character Set ID in hex.

A few things to note when editing a character set definition file:

- You should not remap existing characters.
- All character mappings must be unique.
- New characters should be mapped into the Unicode private use range: e000-f4ff. (Note that the actual Unicode 3.0 private use range is e000-f8ff. However, Oracle reserves f500-f8ff for its own private use.)
- No line in the character set definition file can be longer than 80 characters.

If a character set is derived from an existing Oracle character set, Oracle Corporation recommends using the following character set naming convention:

`<Oracle_character_set_name><organization_name>EXT<version>`

For example, if a company such as Sun Microsystems were adding user-defined characters to the JA16EUC character set, the following character set name might be appropriate:

`JA16EUCSUNWEXT1`

where:

- **JA16EUC**
Is the character set name defined by Oracle
- **SUNW**
Represents the organization name (company stock trading abbreviation for Sun Microsystems)
- **EXT**
Specifies that this is an extension to the JA16EUC character set
- **1**
Specifies the version

Locale Builder Character Set Scenario

This section show how to create a new character set called `MYCHARSET` and use 10001 for its recommended ID number. The scenario will start with an ASCII

character set and add 10 Chinese characters. First, open US7ASCII from the Existing Definitions dialog box. [Figure 11–15](#) illustrates how to begin.

Figure 11–15 Character Set General Information

The screenshot shows the Oracle Locale Builder Utility interface. The 'General' tab is selected, showing fields for 'Character Set Name' (MYCHARSET), 'Character Set ID' (10001), 'ISO Character Set ID' (empty), and 'Base Character Set ID' (empty). A 'Show Existing Definitions...' button is at the bottom. The status bar at the bottom indicates 'Filename: lx20001.nlb', 'Locale Category: Character Set', and 'Status: Editing'.

Field	Value
Character Set Name:	MYCHARSET
Character Set ID:	10001
ISO Character Set ID:	
Base Character Set ID:	

Buttons: Show Existing Definitions...

Status Bar: Filename: lx20001.nlb | Locale Category: Character Set | Status: Editing

In [Figure 11–15](#), the ISO Character Set ID and Base Character Set ID fields are optional. The Base Character Set ID is used for inheriting values so that the base character set's properties are used as a starting template. The Character Set ID is automatically generated, although you can override it. The valid range for a user-defined character set ID is 10,000 to 20,000.

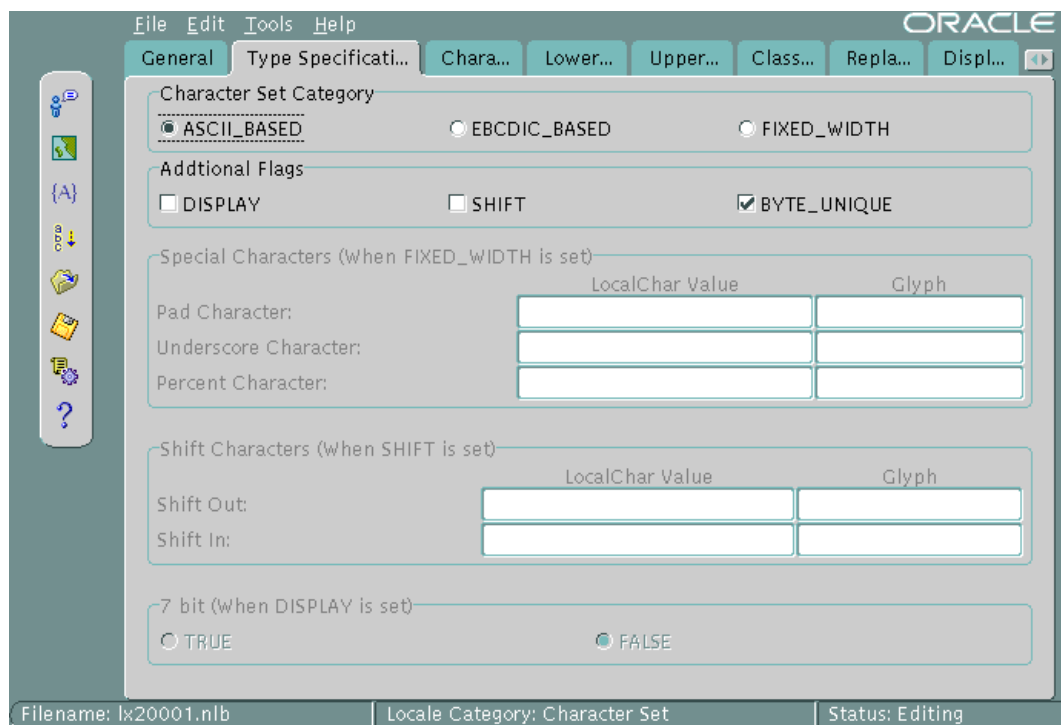
Figure 11–16 Character Set Type Specifications

Figure 11–16 illustrates how to change certain character set specifications. This should not normally be necessary.

When you open a character set, all possible settings for this tab should already be set to appropriate settings. You should keep these settings unless you have a specific reason for changing them. If you need to change the settings, use the following guidelines:

- **FIXED_WIDTH** is to identify character sets whose characters have a uniform length. AL16UTF16 is one example.
- **BYTE_UNIQUE** means the single byte range of codepoints is distinct from multibyte range. An example is JA16EUC.
- **DISPLAY** identifies character sets that have certain character mode characteristics. Arabic and Devanagari character sets are examples.

- SHIFT is for certain character sets that require extra shift characters to distinguish between single-byte characters and multibyte characters.

See Also: [Chapter 2, "Choosing a Character Set"](#) for more information about SHIFT In SHIFT Out character sets

Figure 11–17 Character Set User-Defined General Information

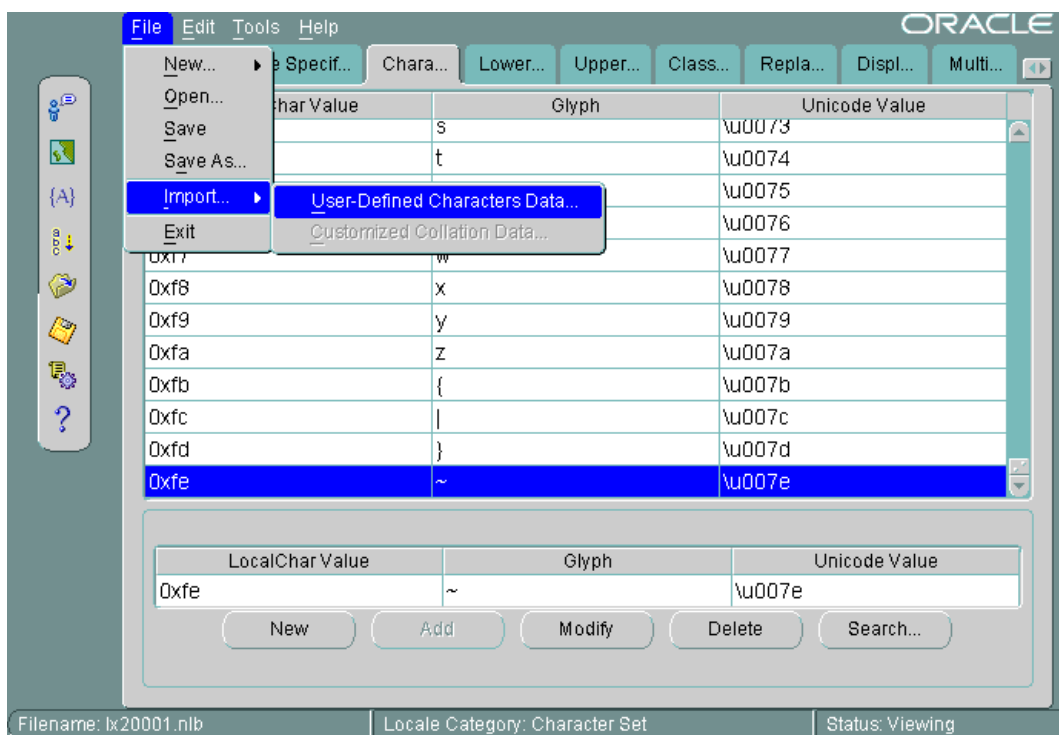


Figure 11–17 illustrates how to add user-defined characters. In this case, you can add characters after 0xfe. You can add one character at a time or use a text file to import a large number of characters. In this example, we first import a file containing the following characters:

```
88a2 963f
88a3 54c0
88a4 611b
88a5 6328
88a6 59f6
```

88a7 9022
 88a8 8475
 88a9 831c
 88aa 7a50
 88ab 60aa

Figure 11–18 Character Set Characters

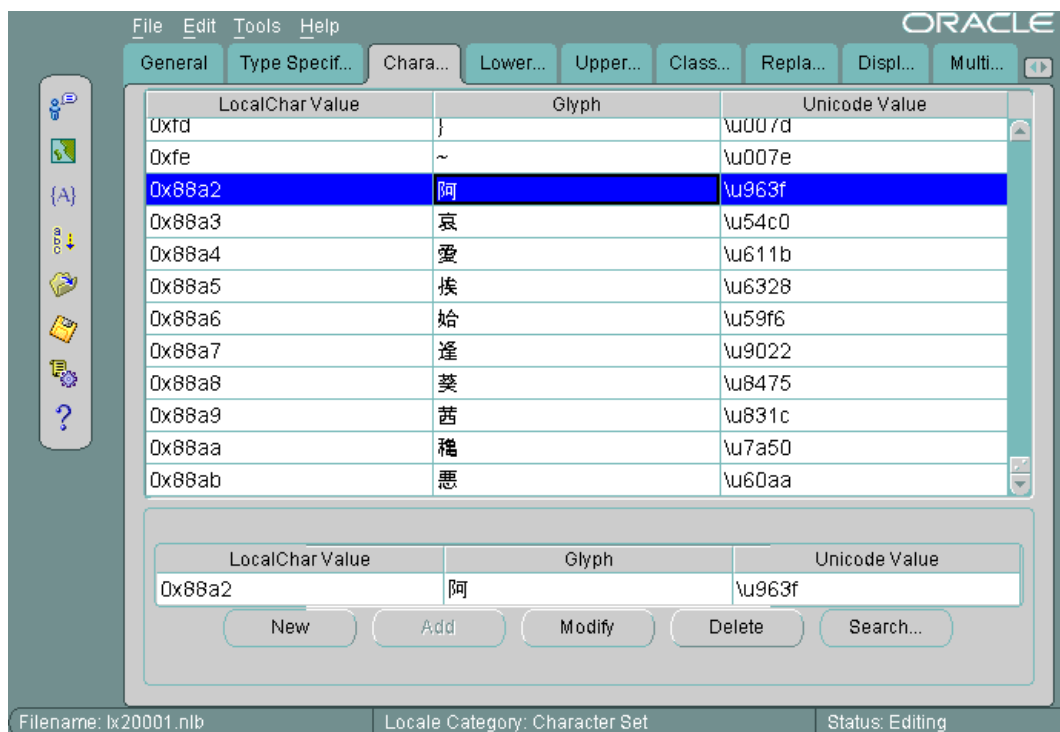
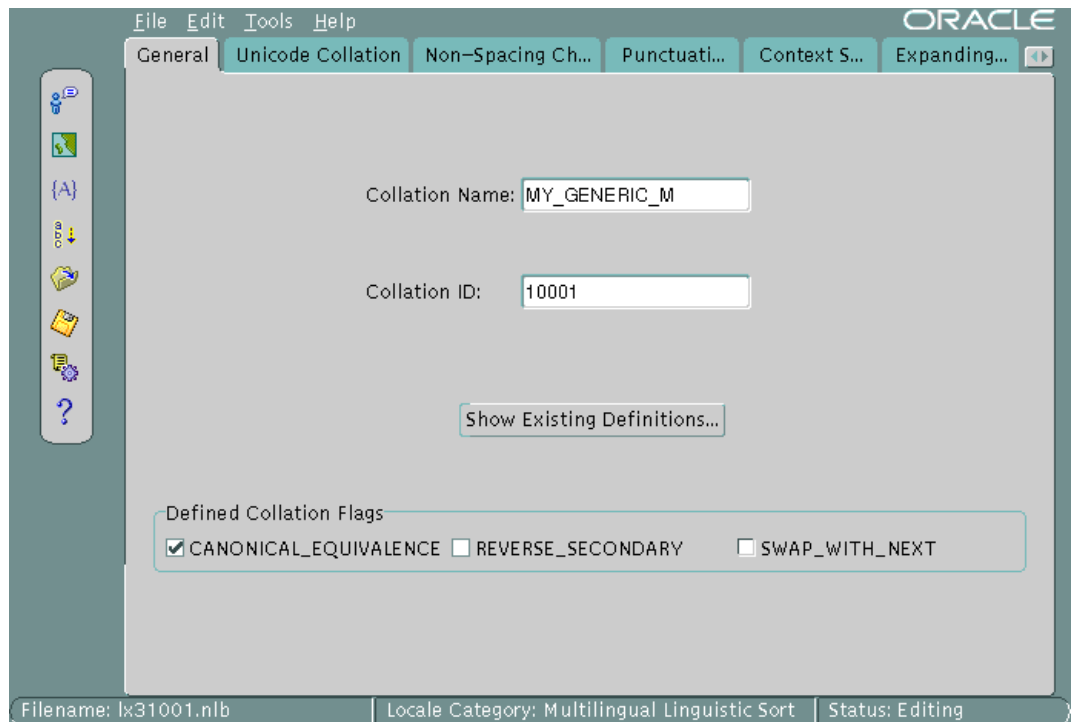


Figure 11–18 illustrates the new characters added after 0xfe. We imported the characters in this case from a file having two columns, with the left column being the local code value and the right column being its Unicode mapping.

Sorting with the Locale Builder

This section shows how to create a new multilingual linguistic sort called `MY_GENERIC_M`, and use 10001 for its ID number. The choice of sort name is based on the convention `GENERIC_M` representing a multilingual ISO sort. In this case, we use `GENERIC_M` as a starting point. [Figure 11–15](#) illustrates how to begin.

Figure 11–19 Collation General Information

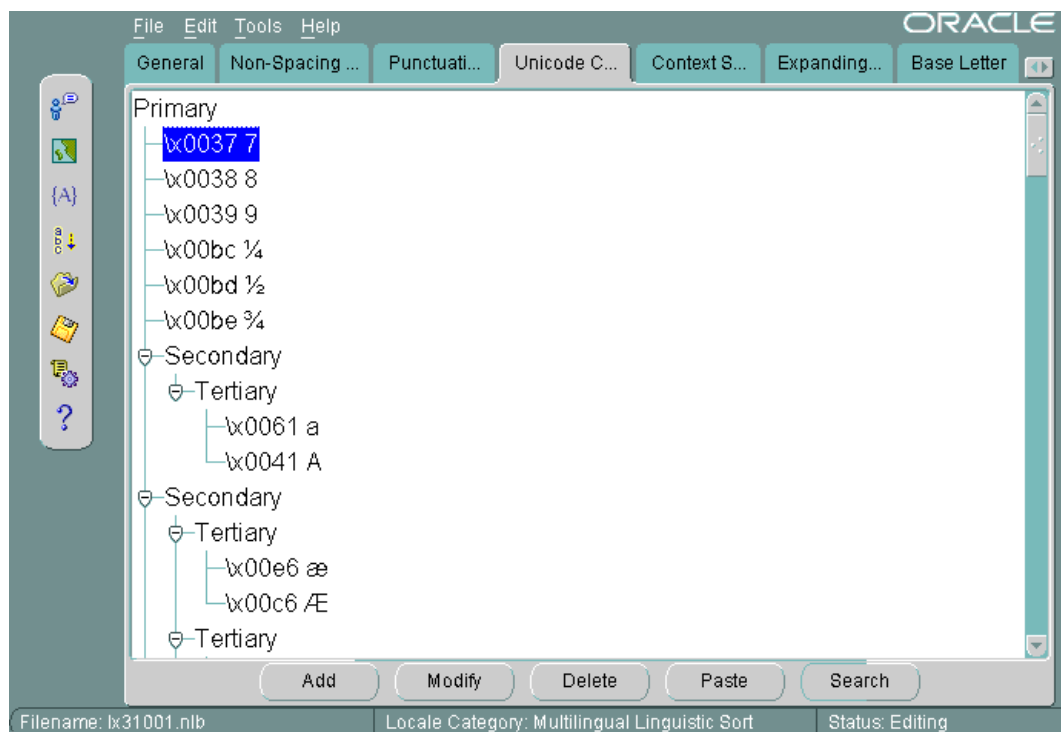


Typical settings for the flags are automatically derived. `SWAP_WITH_NEXT` is relevant for Thai and Lao sorts. `REVERSE_SECONDARY` is for French sorts. `CANONICAL_EQUIVALENCE` determines whether canonical rules will be used.

Collation ID (sort ID) valid ranges for a user-defined sort are 1,000 to 2,000 for monolingual collation and 10,000 to 11,000 for multilingual collation.

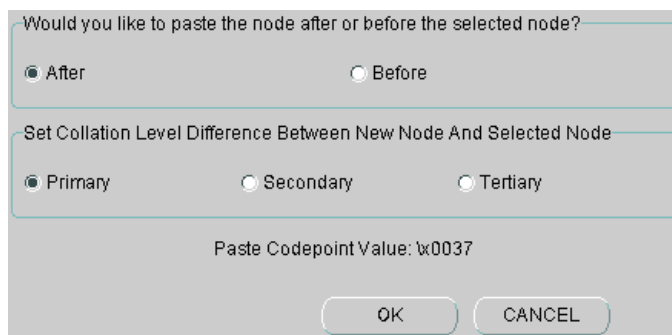
See Also:

- [Figure 11-23, "Collation-Canonical Rules"](#) for more information about canonical rules
- [Chapter 4, "Linguistic Sorting"](#)

Figure 11-20 Collation Unicode Collation

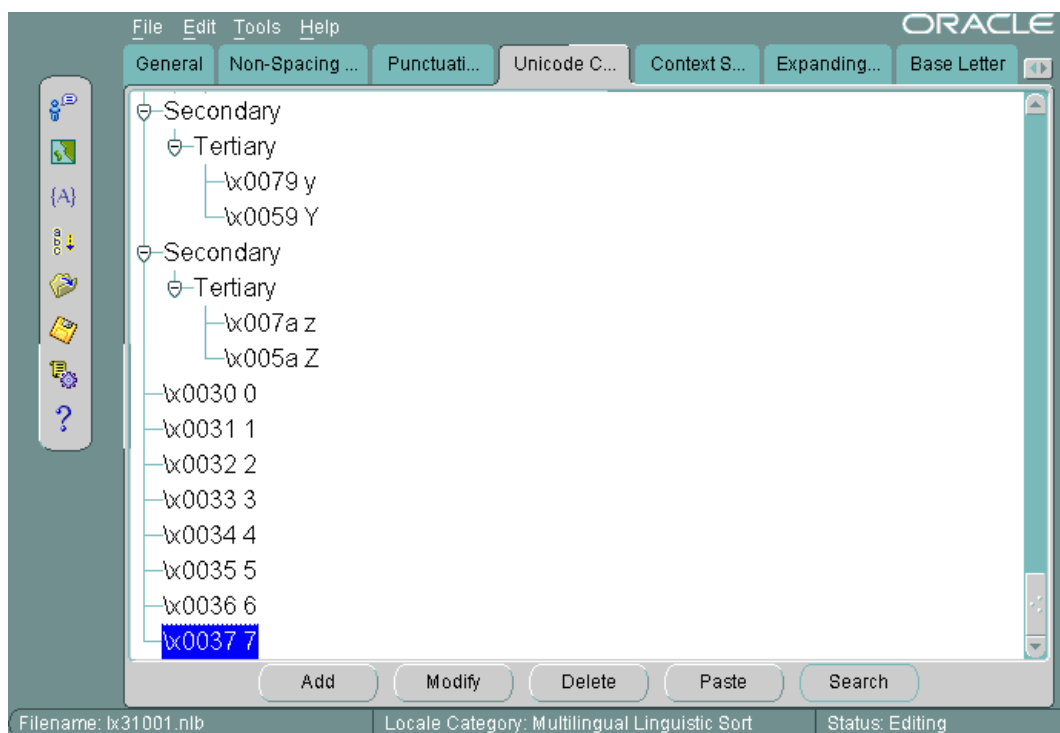
In this scenario, we will move digits so they sort after letters. To do this, we will delete their codepoint values and paste them after the codepoint values of the letters.

[Figure 11-20](#) illustrates selecting a value. Click Delete and paste the value where you want it. Clicking Paste brings up the Collation Pasting Dialog Box, shown in [Figure 11-21](#).

Figure 11–21 Collation Pasting Dialog Box

The dialog box is titled "Collation Pasting Dialog Box". It contains two sections. The first section is titled "Would you like to paste the node after or before the selected node?" and has two radio buttons: "After" (selected) and "Before". The second section is titled "Set Collation Level Difference Between New Node And Selected Node" and has three radio buttons: "Primary" (selected), "Secondary", and "Tertiary". Below these sections is a text field labeled "Paste Codepoint Value: \x0037". At the bottom are two buttons: "OK" and "CANCEL".

In [Figure 11–21](#), choose where to put the deleted node and at what sort level you want it.

Figure 11–22 Collation Unicode Collation After Pasting

In [Figure 11–22](#), we selected the digits 0-7 were moved from their original place before letters a-z to a place after the letters a-z. For multibyte linguistic sorts, the Locale Builder cannot display accented characters, but you can change their sort order.

Changing the Sort Order for Accented Characters

The next scenario is to change the sort order for accented characters. You can do this by changing the sort for all characters containing a particular accent mark or by changing one character at a time. In this example, we change the sort of all characters with a circumflex (for example, û) to go after all characters containing a tilde.

First, we verify the current sort order by choosing Canonical Rules under the Tools menu. This brings up the Canonical Rules dialog box, illustrated in [Figure 11–23](#).

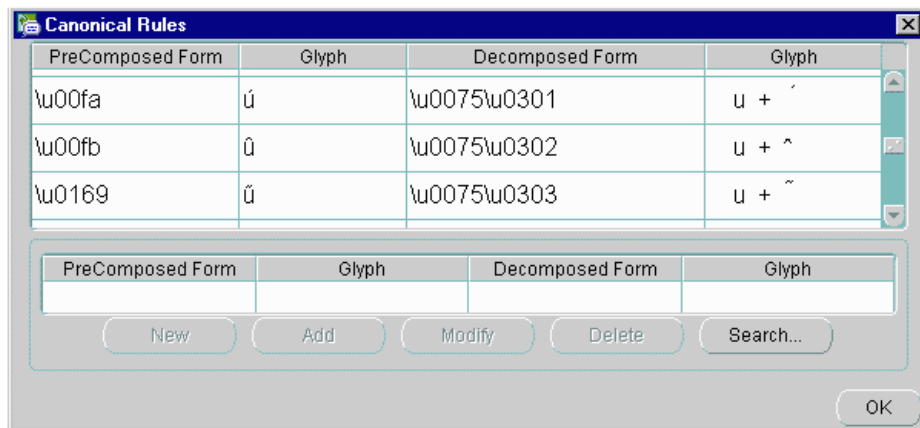
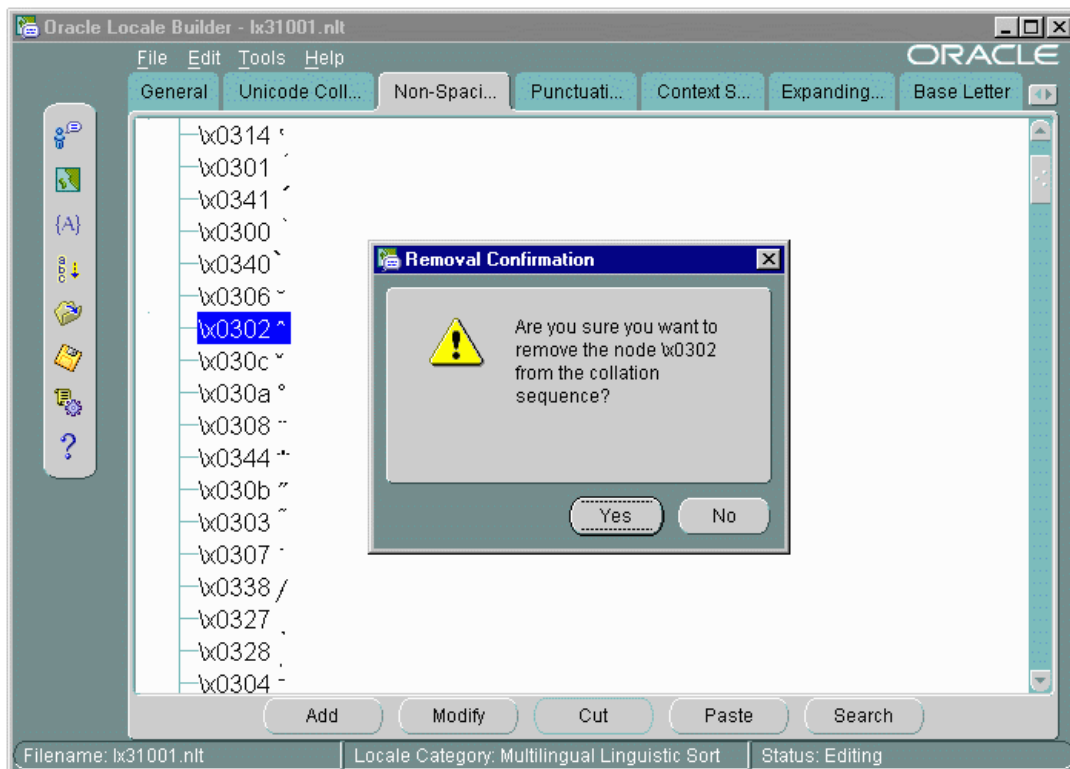
Figure 11–23 Collation-Canonical Rules

Figure 11–23 illustrates how characters are decomposed into their canonical equivalents and their current sorting orders. For example, ã is represented as a plus an umlaut. In this case, we change the sort for all characters with a circumflex so they follow characters with tildes. This example uses a base character of u.

See Also: [Chapter 4, "Linguistic Sorting"](#) for more information about canonical rules

Click on the Non-Spacing tab. If you use the Non-Spacing tab, changes for accent marks apply to all characters.

Figure 11-24 Collation-Changing Several Characters

After selecting the circumflex, click Cut and accept the confirmation. Then all characters with a circumflex will have their sort order changed.

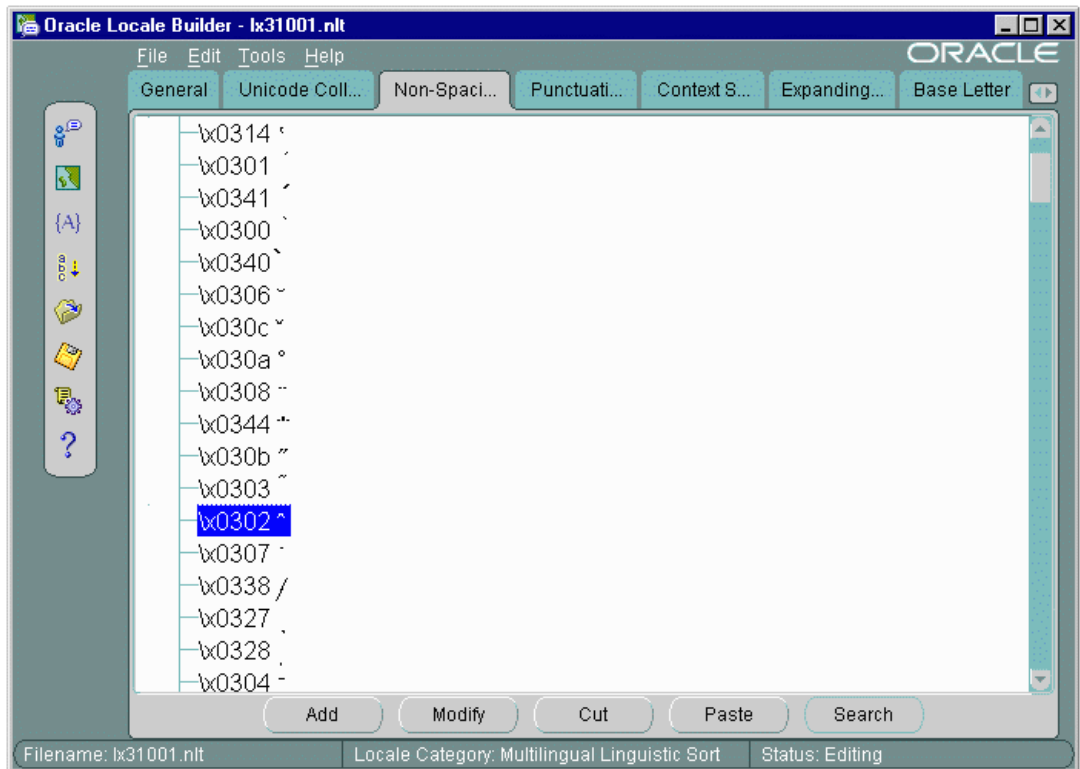
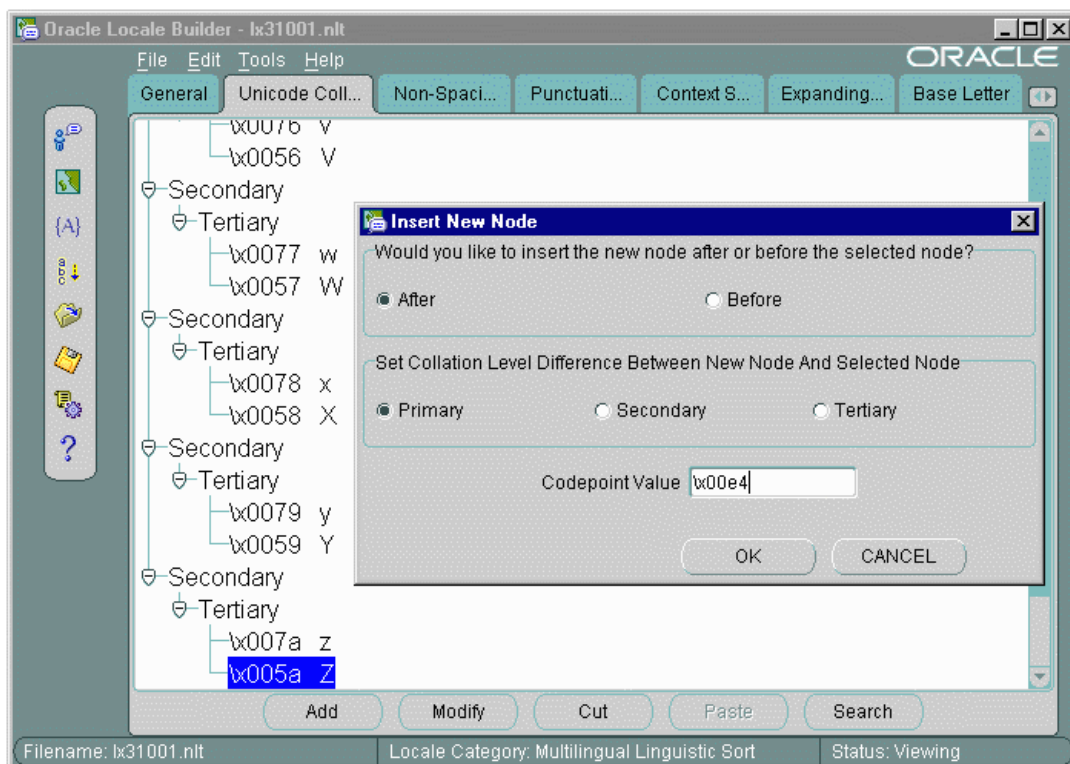
Figure 11–25 Collation-Changing Several Characters

Figure 11–25 illustrates the new order.

Changing the Sort Order for One Accented Character

To change the order of a specific accented character, you need to insert the character directly into the appropriate order position. In this scenario, we will change the sort order for ä so that it sorts after z. First, we select the Unicode Collation tab. Next, we highlight the character next to the one we want, z in this case. Finally, we click Add, which brings up a Paste dialog box.

Figure 11-26 Collation-Changing One Character

As illustrated in [Figure 11-26](#), we choose After and Primary and manually type in `\x00e4`, which is the code point for ä.

We chose Primary for the level because that is the Unicode standard for differentiating between characters having different base letters. A Secondary or Tertiary level sort would also have the same practical results.

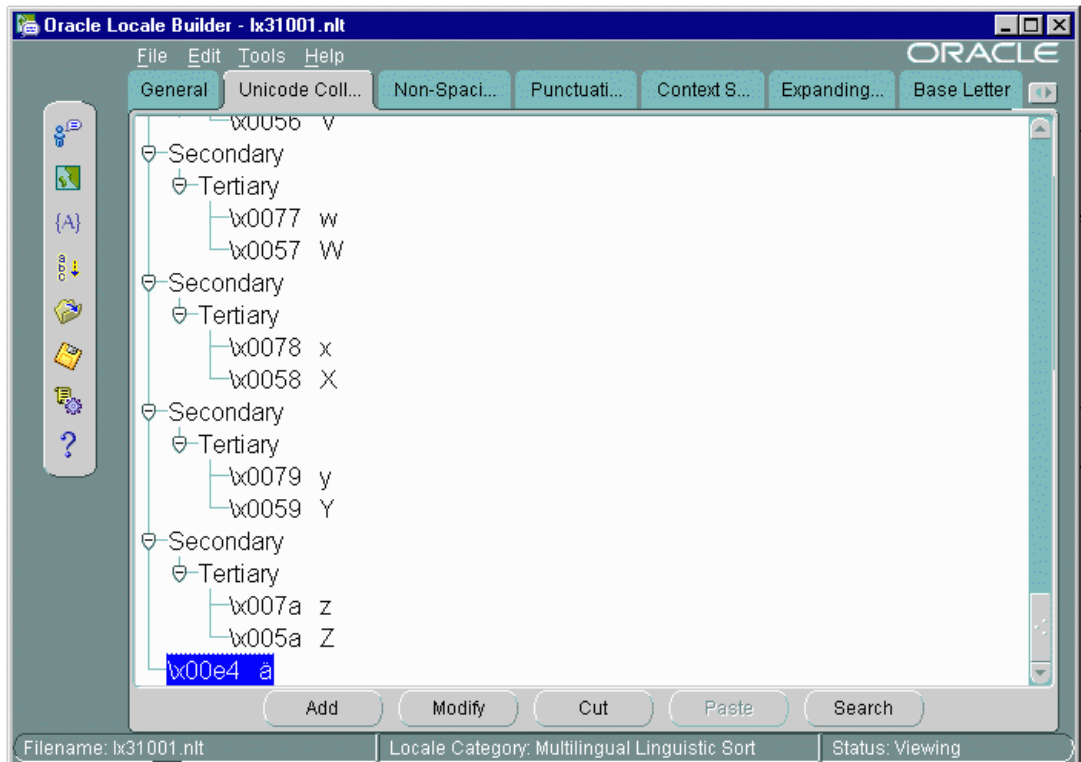
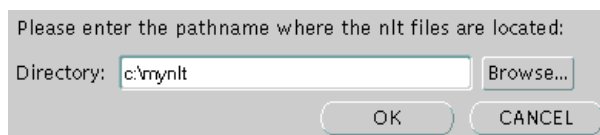
Figure 11–27 Collation-Changing a Single Character

Figure 11–27 shows the final result, and displays the ä correctly.

Generating NLB Files

After you have defined a new language, territory, character set, or linguistic sort, generate new NLB files from the NLT files:

1. Choose Tools > Generate NLB or click the Generate NLB icon in the left side bar.
2. Click Browse to find the directory where the NLT file is located. The location dialog box is shown in Figure 11–28.

Figure 11–28 Generate NLB File

Do not try to specify an NLT file. Oracle Locale Builder generates an NLB file for each NLT file.

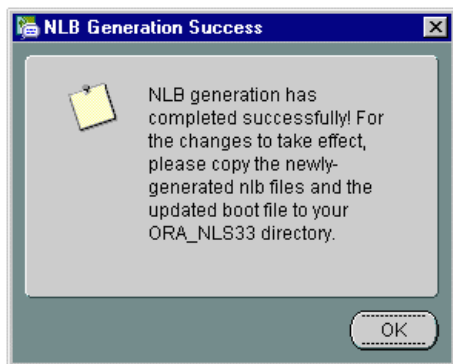
3. Click OK to generate the NLB files.

Using the New NLB Files

The new NLB files do not take effect until you perform the following steps:

1. Copy the NLB files and the `lxlboot.nlb` file into the path that is specified by the `ORA_NLS33` initialization parameter, typically `$ORACLE_HOME/OCOMMON/nls/admin/data`.
2. Restart the database.

[Figure 11–29](#) illustrates the final notification that you have successfully generated NLB files for all NLT files in the directory.

Figure 11–29 NLB Generation Confirmation

Customizing Locale Data

A set of NLS data objects is included with every Oracle distribution set, some of which is customizable. This chapter illustrates how to customize these data objects. It contains the following topics:

- [Customizing Character Sets](#)
- [Customizing Time Zone Data](#)
- [Customizing Calendars](#)
- [NLS Data Installation Utility](#)

Customizing Character Sets

After a locale definition file is created using the Locale Builder, it must be compiled into platform-specific binary files that can be dynamically loaded into memory at runtime. The NLS Data Installation Utility (`lxinst`) described in this chapter allows you to convert and install locale definition text files into binary format, and merge it into an NLS data object set.

Character Set Customization Example

This section uses an example to introduce the steps required to create a new character set. For this example, we will create a new character set based on Oracle's JA16EUC character set and add a few user-defined characters.

Be aware of the following limitations:

- Input of user-defined characters must still be managed by the system, either through an input method or a virtual keyboard.
- Display of user-defined characters must still be managed by the system or the application. In the case of display, a new font specification may be needed. Many vendors provide support of a font editor. After a new font is created, it must be installed onto your system and made accessible to application programs.

See Also: [Chapter 11, "Oracle Locale Builder Utility"](#) for more information about how to customize a character set definition file

Step 1. Back up the NLS binary boot files

Oracle recommends that you backup the NLS installation boot file (`lx0boot.nlb`) and the NLS system boot file (`lx1boot.nlb`) in the `ORA_NLS33` directory before generating and installing NLB files. Enter the following commands:

```
% cd $ORA_NLS33
% cp lx0boot.nlb lx0boot.nlb.orig
% cp lx1boot.nlb lx1boot.nlb.orig
```

Step 2. Generate and install the NLB files

Now you are ready to generate and install the new NLB files. The NLB files are platform-dependent, so regenerate them on each platform and also install these files on both the server and clients.

Use the `lxinst` utility or the Locale Builder to create both the binary character definition files (`lx2ddd.nlb`) and update the NLS boot file (`lx*boot.nlb`).

Example 12-1 lxinst Example

The `lxinst` utility uses the existing system boot file. Therefore, copy the existing binary system boot file into the directory specified by `SYSDIR`. For this example, define `SYSDIR` to be the working directory (`/tmp`). Enter the following command:

```
% cp lx1boot.nlb /tmp
```

The new character set definition file (`lx22710.nlt`) and the text boot file containing the new character set entry (`lx0boot.nlt`) that was created in Steps 2 and 3 should reside in the directory specified by `ORANLS`. For this example, specify it to be `/tmp`. Also, since we define `BASE_CHAR_SET` (the base definition file) to be `JA16EUC` (Id 830 in hex value 033e), its NLT file (`lx2033e.nlt`) or NLB file (`lx*033e.nlb`) should also be in the directory specified by `ORANLS`, so that the new character set can inherit all definitions from it. Enter one of the following commands:

```
% cp lx2033e.nlt /tmp
```

or

```
% cp lx*033e.nlb /tmp
```

Use the `lxinst` utility to generate an NLB file (`lx22710.nlb`) for the character set in the directory specified by the `ORANLS` and an updated binary boot file (`lx1boot.nlb`) in the directory specified by `DESTDIR`. For this example, define `ORANLS`, `SYSDIR`, and `DESTDIR` all to be `/tmp`. Enter the following command:

```
% $ORACLE_HOME/bin/lxinst oranls=/tmp sysdir=/tmp destdir=/tmp
```

Install the newly generated binary boot file (`lx1boot.nlb`) into the `ORA_NLS33` directory:

```
% cp /tmp/lx1boot.nlb $ORA_NLS33/lx1boot.nlb
```

Finally, install the new character set definition file (`lx2*.nlb`) into the `ORA_NLS33` directory. If there are files with names similar to `lx5*.nlb` or `lx6*.nlb`, then install them, too:

```
% cp /tmp/lx22710.nlb $ORA_NLS33
```

```
% cp /tmp/lx52710.nlb $ORA_NLS33
```

```
% cp /tmp/lx62710.nlb $ORA_NLS33
```

Step 3. Repeat for Each Platform

You must repeat Step 2 on each hardware platform because the NLB file is a platform-specific binary. It must also be repeated for every system that must recognize the new character set. Therefore, you should compile and install the new NLB files on both server and client machines.

Step 4. Create the Database Using New Character Set

After installing the NLB files, shut down and restart the database server in order to initialize NLS data loading.

After bringing the database server back up, create the new database using the newly created character set.

To use the new character set on the client side, simply exit the client (such as Enterprise Manager or SQL*Plus) and re-invoke it after installing the NLB files.

Using User-Defined Character Sets and Java

Creating Character Set Definition Files

If you have any Java products (for example, JDBC or SQLJ) in your applications and want them to support user-defined characters, you must generate and install a special Java zip file (`gss_custom.zip`) into your Oracle Home directory. The installation steps are as follows:

On UNIX:

```
$ORACLE_HOME/JRE/bin/jre -classpath $ORACLE_HOME/jlib/gss-1_1.zip:$ORACLE_
HOME/jlib/gss_charset-1_2.zip Ginstall <lx22710>.nlt
```

On Windows:

```
%JREHOME%\bin\jre.exe -classpath %ORACLE_HOME%\jlib\gss-1_1.zip;%ORACLE_
HOME%\jlib\gss_charset-1_2.zip Ginstall <lx22710>.nlt
```

where `%JREHOME%` is the `C:\Program Files\Oracle\jre\version_num` directory.

`lx22710.nlt` is an example of an user-defined character set created using the Oracle Locale Builder.

The above commands generate a `gss_custom.zip` file in the current directory. If you need to add support for more than one user-defined character set, you can

append their definitions to the same `gss_custom.zip` file by re-issuing the above command for each of the additional user-defined character sets. For example:

```
$ORACLE_HOME/JRE/bin/jre -classpath $ORACLE_HOME/jlib/gss-1_1.zip:
    $ORACLE_HOME/jlib/gss_charset-1_2.zip Ginstall <lx22710>.nlt
```

```
$ORACLE_HOME/JRE/bin/jre -classpath $ORACLE_HOME/jlib/gss-1_1.zip:
    $ORACLE_HOME/jlib/gss_charset-1_2.zip Ginstall <lx22711>.nlt
```

```
$ORACLE_HOME/JRE/bin/jre -classpath $ORACLE_HOME/jlib/gss-1_1.zip:
    $ORACLE_HOME/jlib/gss_charset-1_2.zip Ginstall <lx22712>.nlt
```

`lx22710.nlt`, `lx22711.nlt` and `lx22712.nlt` will all be contained in `gss_custom.zip`.

After `gss_custom.zip` has been created, store it in the `$ORACLE_HOME/ocommon/nls/admin/data` directory. For example:

```
% cp gss_custom.zip $ORACLE_HOME/ocommon/nls/admin/data
```

Enabling Java and User-Defined Character Sets

There are three Java components where you may want to add the `gss_custom.zip` file:

- Java Virtual Machine (for Java in the database)

On UNIX:

```
%loadjava -u sys/<passwd> -grant EXECUTE -synonym -r -r -v gss_custom.zip
```

On Windows:

```
loadjava -u sys/<passwd> -grant EXECUTE -synonym -r -r -v gss_custom.zip
```

This loads it into the database. Note that `<password>` needs to be replaced by the password for SYS.

- Apache (for using servlets)

Edit the file `jserv.properties` as follows:

On UNIX, add the line:

```
wrapper.classpath = $ORACLE_HOME/ocommon/nls/admin/data/gss_custom.zip
```

On Windows, add the line:

```
wrapper.classpath = %ORA_HOME%\ocommon\nls\admin\data\gss_custom.zip
```

- Client environment (for JDBC on the client)

On UNIX:

```
%set CLASSPATH $ORACLE_HOME/ocommon/nls/admin/data/gss_custom.zip
```

On Windows:

Add the path %ORACLE_HOME%\ocommon\nls\admin\data\gss_custom.zip to your existing CLASSPATH.

See Also: [Chapter 11, "Oracle Locale Builder Utility"](#) for more information about user-defined character sets

Customizing Time Zone Data

The time zone files contain the valid time zone names. The following information is included for each zone:

- Offset from UTC (Coordinated Universal Time)
- Transition times for daylight savings
- Abbreviations for standard time and daylight savings time

Abbreviations are only used in conjunction with the zone names. There are 2 timezone files under the Oracle installation directory (ORACLE_HOME):

- oracore/zoneinfo/timzone.dat

This is the default. It contains the most commonly used time zones and is smaller for better database performance.

- oracore/zoneinfo/timezlg.dat

This file contains the larger set of defined time zones and should be used by customers who require time zones that are not defined in the default timzone.dat file. This larger set of time zone information might affect performance.

To enable the use of the larger time zone data file, you must:

1. Shut down the database.
2. Set the ORA_TZFILE environment variable to the full pathname of the location for the timezlg.dat file.

3. Restart the database.

After the larger `timezlg.dat` is used, it must continue to be used unless the user is sure that none of the non-default zones are used for data that is stored in the database. Also, all databases that share information must use the same timezone data file.

To view the timezone names, issue the following query:

```
SELECT * FROM V$TIMEZONE_NAMES;
```

Customizing Calendars

A number of calendars besides Gregorian are supported. Although all of them are defined with data linked directly into NLS, some of them may require the addition of ruler eras (in the case of imperial calendars) or deviation days (in the case of lunar calendars) in the future. In order to do this without waiting for a new release, you can define the additional eras or deviation days in an external file, which is then automatically loaded when executing the calendar functions.

The calendar data is first defined in a text-format definition file. This file must be converted into binary format before it can be used. The Calendar Utility described here allows you to do this.

NLS Calendar Utility

Usage

The Calendar Utility takes as input a text-format definition file. The name of the file and its location are hard-coded as a platform-dependent value. On UNIX platforms, the file name is `lxecal.nlb`, and its location is `$ORACLE_HOME/ocommon/nls`. A sample calendar definition file is included in the distribution.

Note: The location of files is platform-dependent. See your platform-specific Oracle documentation for information about the location of files on your system.

The `lxegen` executable produces as output a binary file containing the calendar data in the appropriate format. The name of the output file is also hard-coded as a platform-dependent value. On UNIX, the name is `lxecal.nlb`. The file will be

generated in the same directory as the text-format file, and an already-existing file will be overwritten.

Once the binary file has been generated, it will automatically be loaded during system initialization. Do not move or rename the file, as it is expected to be found in the same hard-coded name and location.

Syntax

The Calendar Utility is invoked directly from the command line:

`LXEGEN`

There are no parameters.

NLS Data Installation Utility

When you order an Oracle distribution set, a default set of NLS data objects is included. Some NLS data objects are customizable. For example, in Oracle9i, you can extend Oracle's character set definition files to add user-defined characters. These NLS definition files must be converted into binary format and merged into the existing NLS object set. The NLS Data Installation Utility allows you to do this.

Along with the binary object files, a boot file is generated by the NLS Data Installation Utility. This boot file is used by the modules to identify and locate all the NLS objects which it needs to load.

To facilitate boot file distribution and user configuration, three types of boot files are defined:

Installation Boot File	The boot file included as part of the distribution set.
System Boot File	The boot file generated by the NLS Data Installation Utility which loads the NLS objects. If the user already has an installed system boot file, its contents can be merged with the new system boot file during object generation.
User Boot File	A boot file that contains a subset of the system boot file information.

Syntax

The NLS Data Installation Utility is invoked from the command line with the following syntax:

`LXINST [ORANLS=pathname] [SYSDIR=pathname] [DESTDIR=pathname] [HELP=[yes | no]]`

[WARNING=[0 | 1 | 2 | 3]]

where

ORANLS= <i>pathname</i>	Specifies where to find the text-format boot and object files and where to store the new binary-format boot and object files. If not specified, NLS Installation Utility uses the value in the environment variable ORA_NLS33 (or the equivalent for your operating system). If both are specified, the command line parameter overrides the environment variable. If neither is specified, the NLS Installation Utility will exit with an error.
SYSDIR= <i>pathname</i>	Specifies where to find the existing system boot file. If not specified, the NLS Installation Utility uses the directory specified in the initialization file parameter ORANLS. If there is no existing system boot file or the NLS Installation Utility is unable to find the file, it will create a new file and copy it to the appropriate directory.
DESTDIR= <i>pathname</i>	Specifies where to put the new (merged) system boot file. If not specified, the NLS Installation Utility uses the directory specified in the initialization file parameter ORANLS. Any system boot file that exists in this directory will be overwritten, so make a backup first.
HELP=[yes no]	If "yes", a help message describing the syntax for the NLS Installation Utility will be displayed.
[WARNING= 0 1 2 3]]	If you specify "0", no warning messages are displayed. If you specify "1", all messages for level 1 will be displayed. If you specify "2", all messages for levels 2 and 1 will be displayed. If you specify "3", all messages for levels 3, 2, and 1 will be displayed.

Return Codes

You may receive the following return codes upon executing `lxinst`:

0	The generation of the binary boot and object files, and merge of the installation and system boot files completed successfully.
1	Installation failed: the NLS Installation Utility will exit with an error message that describes the problem.

Usage

Use `lxinst` to install customized character sets by completing the following tasks:

- Create a text-format boot file (`lx0boot.nlt`) containing references to new data objects.
 - Data objects can be generated only if they are referenced in the boot file.
 - You can generate only character set object types.
- Create your new text-format data object files.

See Also: ["Data Object File Names"](#) on page 12-11 for naming conventions

Note: Your distribution set contains a character set definition demonstration file that you can use as a reference or as a template. On UNIX-based systems, this file is located in `$ORACLE_HOME/demo/*.nlt`

- Invoke `lxinst` as described above (using the appropriate parameters) to generate new binary data object files. These files will be generated in the directory you specified in `ORANLS`.
 - `lxinst` also generates both a new installation boot file and system boot file. If you have a previous NLS installation and want to merge the existing information with the new in the system boot file, copy the existing system boot file into the directory you specified in `SYSDIR`. A new system boot file containing the merged information is generated in the directory specified in `DESTDIR`.

Note: As always, you should have backups of any existing files you do not want overwritten.

Object Types

Only character set object types are currently supported for customizing.

Object IDs

NLS data objects are uniquely identified by a numeric object ID. The ID may never have a zero or negative value.

In general, you can define new objects as long as you specify the object ID within the range 10000-20000.

Object Names

Only a very restricted set of characters can be used in object names:

ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_- and <space>

Object names must start with an alphabetic character. Language, territory, and character set names cannot contain an underscore character, but linguistic definition names can. There is no case distinction in object names, and the maximum size of an object name is 30 bytes (excluding terminating null).

Data Object File Names

The system-independent object file name is constructed from the generic boot file entry information:

1xtddd

where:

t	1 digit object type (hex)
ddd	4 digit object ID (hex)

The installation boot file name is 1x0BOOT; the system boot file name is 1x1BOOT; user boot files are named 1x2BOOT. The file extension for text format files is .nlt. The file extension for binary files is .nlb.

Examples:

lx22711.nlt	Text-format character set definition, ID=10001
lx0boot.nlt	Text-format installation boot file
lx1boot.nlb	Binary system boot file
lx22711.nlb	Binary character set definition, ID=10001

Locale Data

This appendix lists the languages, territories, character sets, and other locale data supported by the Oracle server. It includes these topics:

- [Languages](#)
- [Translated Messages](#)
- [Territories](#)
- [Character Sets](#)
- [Linguistic Sorting](#)
- [Calendar Systems](#)
- [Obsolete Locale Data](#)

You can also obtain information about supported character sets, languages, territories, and sorting orders by querying the dynamic data view `V$NLS_VALID_VALUES`.

See Also: *Oracle9i Database Reference* for more information about the data that can be returned by this view

Languages

[Table A-1](#) lists the languages supported by the Oracle server.

Table A-1 *Oracle Supported Languages*

Name	Abbreviation
AMERICAN	us
ARABIC	ar
ASSAMESE	as
BANGLA	bn
BENGALI	bn
BRAZILIAN PORTUGUESE	ptb
BULGARIAN	bg
CANADIAN FRENCH	fr
CATALAN	ca
CROATIAN	hr
CZECH	cs
DANISH	dk
DUTCH	nl
EGYPTIAN	eg
ENGLISH	gb
ESTONIAN	et
FINNISH	sf
FRENCH	f
GERMAN DIN	din
GERMAN	d
GREEK	el
GUJARATI	gu
HEBREW	iw
HINDI	hi

Table A-1 Oracle Supported Languages (Cont.)

Name	Abbreviation
HUNGARIAN	hu
ICELANDIC	is
INDONESIAN	in
ITALIAN	i
JAPANESE	ja
KANNADA	kn
KOREAN	ko
LATIN AMERICAN SPANISH	esa
LATVIAN	lv
LITHUANIAN	lt
MALAY	ms
MALAYALAM	ml
MARATHI	mr
MEXICAN SPANISH	esm
NORWEGIAN	n
ORIYA	or
POLISH	pl
PORTUGUESE	pt
PUNJABI	pa
ROMANIAN	ro
RUSSIAN	ru
SIMPLIFIEDD CHINESE	zhs
SLOVAK	sk
SLOVENIAN	sl
SPANISH	e
SWEDISH	s
TAMIL	ta

Table A-1 Oracle Supported Languages (Cont.)

Name	Abbreviation
TELUGU	te
THAI	th
TRADITIONAL CHINESE	zht
TURKISH	tr
UKRAINIAN	uk
VIETNAMESE	vn

Translated Messages

Oracle error messages have been translated into the languages which are listed in [Table A-2](#).

Table A-2 Oracle Supported Messages

Name	Abbreviation
ARABIC	ar
BRAZILIAN PORTUGUESE	ptb
CATALAN	ca
CZECH	cs
DANISH	dk
DUTCH	nl
FINNISH	sf
FRENCH	f
GERMAN	d
GREEK	el
HEBREW	iw
HUNGARIAN	hu
ITALIAN	i
JAPANESE	ja
KOREAN	ko

Table A–2 Oracle Supported Messages (Cont.)

Name	Abbreviation
LATIN AMERICAN SPANISH	esa
NORWEGIAN	n
POLISH	pl
PORTUGUESE	pt
ROMANIAN	ro
RUSSIAN	ru
SIMPLIFIED CHINESE	zhs
SLOVAK	sk
SPANISH	e
SWEDISH	s
TRADITIONAL CHINESE	zht
TURKISH	tr

Territories

[Table A–3](#) lists the territories supported by the Oracle server.

Table A–3 Oracle Supported Territories

Name		
ALGERIA	HONG KONG	PERU
AMERICA	HUNGARY	POLAND
AUSTRALIA	ICELAND	PORTUGAL
AUSTRIA	INDIA	PUERTO RICO
BAHRAIN	INDONESIA	QATAR
BANGLADESH	IRAQ	ROMANIA
BELGIUM	IRELAND	SAUDI ARABIA
BRAZIL	ISRAEL	SINGAPORE
BULGARIA	ITALY	SLOVAKIA
CANADA	JAPAN	SLOVENIA

Table A–3 Oracle Supported Territories (Cont.)

Name		
CATALONIA	JORDAN	SOMALIA
CHILE	KAZAKHSTAN	SOUTH AFRICA
CHINA	KOREA	SPAIN
CIS	KUWAIT	SUDAN
COLOMBIA	LATVIA	SWEDEN
COSTA RICA	LEBANON	SWITZERLAND
CROATIA	LIBYA	SYRIA
CYPRUS	LITHUANIA	TAIWAN
CZECH REPUBLIC	LUXEMBOURG	THAILAND
DENMARK	MACEDONIA	THE NETHERLANDS
DJIBOUTI	MALAYSIA	TUNISIA
EGYPT	MAURITANIA	TURKEY
EL SALVADOR	MEXICO	UKRAINE
ESTONIA	MOROCCO	UNITED ARAB EMIRATES
FINLAND	NEW ZEALAND	UNITED KINGDOM
FRANCE	NICARAGUA	UZBEKISTAN
GUATEMALA	NORWAY	VENEZUELA
GERMANY	OMAN	VIETNAM
GREECE	PANAMA	YEMEN
		YUGOSLAVIA

Character Sets

Oracle-supported character sets are listed below, for easy reference, according to three broad language groups. In addition, common subset/superset combinations are listed.

- [Asian Language Character Sets](#)
- [European Language Character Sets](#)
- [Middle Eastern Language Character Sets](#)

Note that some character sets may be listed under multiple language groups because they provide multilingual support. For instance, Unicode spans the Asian, European, and Middle Eastern language groups because it supports most of the major scripts of the world.

The comment section indicates the type of encoding used:

SB = Single-byte encoding

MB = Multibyte encoding

FIXED = Fixed-width multi-byte encoding

As mentioned in [Chapter 3, "Setting Up a Globalization Support Environment"](#), the type of encoding affects performance, so use the most efficient encoding that meets your language needs. Also, some encoding types can only be used with certain data types. For instance, the AL16UTF16 character set can only be used as an NCHAR character set, and not as a database character set.

Also documented in the comment section are other unique features of the character set that may be important to users or your database administrator. For instance, whether the character set supports the new Euro currency symbol, whether user-defined characters are supported for character set customization, and whether the character set is a strict superset of ASCII (which will allow you to make use of the ALTER DATABASE [NATIONAL] CHARACTER SET statement in case of migration.)

EURO = Euro symbol supported

UDC = User-defined characters supported

ASCII = Strict superset of ASCII

Oracle does not document individual code page layouts. For specific details about a particular character set, its character repertoire, and code point values, you should refer to the actual national, international, or vendor-specific standards.

Asian Language Character Sets

[Table A-4](#) lists the Oracle character sets that can support Asian languages.

Table A-4 Asian Language Character Sets

Name	Description	Comments
BN8BSCII	Bangladesh National Code 8-bit BSCII	SB, ASCII
ZHT16BIG5	BIG5 16-bit Traditional Chinese	MB, ASCII
ZHT16HKSCS	MS Windows Code Page 950 with Hong Kong Supplementary Character Set	MB, ASCII, EURO
ZHS16CGB231280	CGB2312-80 16-bit Simplified Chinese	MB, ASCII
ZHS32GB18030	GB18030-2000	MB, ASCII, EURO
JA16EUC	EUC 24-bit Japanese	MB, ASCII
JA16EUCYEN	EUC 24-bit Japanese with '\' mapped to the Japanese yen character	MB
ZHT32EUC	EUC 32-bit Traditional Chinese	MB, ASCII
ZHS16GBK	GBK 16-bit Simplified Chinese	MB, ASCII, UDC
ZHT16CCDC	HP CCDC 16-bit Traditional Chinese	MB, ASCII
JA16DBCS	IBM EBCDIC 16-bit Japanese	MB, UDC
JA16EBCDIC930	IBM DBCS Code Page 290 16-bit Japanese	MB, UDC
KO16DBCS	IBM EBCDIC 16-bit Korean	MB, UDC
ZHS16DBCS	IBM EBCDIC 16-bit Simplified Chinese	MB, UDC
ZHT16DBCS	IBM EBCDIC 16-bit Traditional Chinese	MB, UDC
KO16KSC5601	KSC5601 16-bit Korean	MB, ASCII
KO16KSCCS	KSCCS 16-bit Korean	MB, ASCII
JA16VMS	JVMS 16-bit Japanese	MB, ASCII
ZHS16MACCGB231280	Mac client CGB2312-80 16-bit Simplified Chinese	MB
JA16MACSJIS	Mac client Shift-JIS 16-bit Japanese	MB
TH8MACTHAI	Mac Client 8-bit Latin/Thai	SB
TH8MACTHAIS	Mac Server 8-bit Latin/Thai	SB, ASCII
TH8TISEBCDICS	Thai Industrial Standard 620-2533-EBCDIC Server 8-bit	SB

Table A–4 Asian Language Character Sets (Cont.)

Name	Description	Comments
ZHT16MSWIN950	MS Windows Code Page 950 Traditional Chinese	MB, ASCII, UDC
KO16MSWIN949	MS Windows Code Page 949 Korean	MB, ASCII, UDC
VN8MSWIN1258	MS Windows Code Page 1258 8-bit Vietnamese	SB, ASCII, EURO
IN8ISCII	Multiple-Script Indian Standard 8-bit Latin/Indian Languages	SB, ASCII
JA16SJIS	Shift-JIS 16-bit Japanese	MB, ASCII, UDC
JA16SJISYEN	Shift-JIS 16-bit Japanese with '\ ' mapped to the Japanese yen character	MB, UDC
ZHT32SOPS	SOPS 32-bit Traditional Chinese	MB, ASCII
ZHT16DBT	Taiwan Taxation 16-bit Traditional Chinese	MB, ASCII
TH8TISASCII	Thai Industrial Standard 620-2533 - ASCII 8-bit	SB, ASCII, EURO
TH8TISEBCDIC	Thai Industrial Standard 620-2533 - EBCDIC 8-bit	SB
ZHT32TRIS	TRIS 32-bit Traditional Chinese	MB, ASCII
AL16UTF16	See "Universal Character Sets" on page A-18 for details	MB, EURO, FIXED
AL32UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTFE	See "Universal Character Sets" on page A-18 for details	MB, EURO
VN8VN3	VN3 8-bit Vietnamese	SB, ASCII

European Language Character Sets

[Table A–5](#) lists the Oracle character sets that can support European languages.

Table A–5 European Language Character Sets

Name	Description	Comments
US7ASCII	ASCII 7-bit American	SB, ASCII
SF7ASCII	ASCII 7-bit Finnish	SB
YUG7ASCII	ASCII 7-bit Yugoslavian	SB
RU8BESTA	BESTA 8-bit Latin/Cyrillic	SB, ASCII
EL8GCOS7	Bull EBCDIC GCOS7 8-bit Greek	SB

Table A–5 *European Language Character Sets (Cont.)*

Name	Description	Comments
WE8GCOS7	Bull EBCDIC GCOS7 8-bit West European	SB
EL8DEC	DEC 8-bit Latin/Greek	SB
TR7DEC	DEC VT100 7-bit Turkish	SB
TR8DEC	DEC 8-bit Turkish	SB, ASCII
TR8EBCDIC1026	EBCDIC Code Page 1026 8-bit Turkish	SB
TR8EBCDIC1026S	EBCDIC Code Page 1026 Server 8-bit Turkish	SB
TR8PC857	IBM-PC Code Page 857 8-bit Turkish	SB, ASCII
TR8MACTURKISH	MAC Client 8-bit Turkish	SB
TR8MACTURKISHS	MAC Server 8-bit Turkish	SB, ASCII
TR8MSWIN1254	MS Windows Code Page 1254 8-bit Turkish	SB, ASCII, EURO
WE8BS2000L5	Siemens EBCDIC.DFL5 8-bit West European/Turkish	SB
WE8DEC	DEC 8-bit West European	SB, ASCII
D7DEC	DEC VT100 7-bit German	SB
F7DEC	DEC VT100 7-bit French	SB
S7DEC	DEC VT100 7-bit Swedish	SB
E7DEC	DEC VT100 7-bit Spanish	SB
NDK7DEC	DEC VT100 7-bit Norwegian/Danish	SB
I7DEC	DEC VT100 7-bit Italian	SB
NL7DEC	DEC VT100 7-bit Dutch	SB
CH7DEC	DEC VT100 7-bit Swiss (German/French)	SB
SF7DEC	DEC VT100 7-bit Finnish	SB
WE8DG	DG 8-bit West European	SB, ASCII
WE8EBCDIC37C	EBCDIC Code Page 37 8-bit Oracle/c	SB
WE8EBCDIC37	EBCDIC Code Page 37 8-bit West European	SB
D8EBCDIC273	EBCDIC Code Page 273/1 8-bit Austrian German	SB
DK8EBCDIC277	EBCDIC Code Page 277/1 8-bit Danish	SB
S8EBCDIC278	EBCDIC Code Page 278/1 8-bit Swedish	SB

Table A–5 European Language Character Sets (Cont.)

Name	Description	Comments
I8EBCDIC280	EBCDIC Code Page 280/1 8-bit Italian	SB
WE8EBCDIC284	EBCDIC Code Page 284 8-bit Latin American/Spanish	SB
WE8EBCDIC285	EBCDIC Code Page 285 8-bit West European	SB
WE8EBCDIC924	Latin 9 EBCDIC 924	SB, EBCDIC
WE8EBCDIC1047	EBCDIC Code Page 1047 8-bit West European	SB
WE8EBCDIC1047E	Latin 1/Open Systems 1047	SB, EBCDIC, EURO
WE8EBCDIC1140	EBCDIC Code Page 1140 8-bit West European	SB, EURO
WE8EBCDIC1140C	EBCDIC Code Page 1140 Client 8-bit West European	SB, EURO
WE8EBCDIC1145	EBCDIC Code Page 1145 8-bit West European	SB, EURO
WE8EBCDIC1146	EBCDIC Code Page 1146 8-bit West European	SB, EURO
WE8EBCDIC1148	EBCDIC Code Page 1148 8-bit West European	SB, EURO
WE8EBCDIC1148C	EBCDIC Code Page 1148 Client 8-bit West European	SB, EURO
F8EBCDIC297	EBCDIC Code Page 297 8-bit French	SB
WE8EBCDIC500C	EBCDIC Code Page 500 8-bit Oracle/c	SB
WE8EBCDIC500	EBCDIC Code Page 500 8-bit West European	SB
EE8EBCDIC870	EBCDIC Code Page 870 8-bit East European	SB
EE8EBCDIC870C	EBCDIC Code Page 870 Client 8-bit East European	SB
EE8EBCDIC870S	EBCDIC Code Page 870 Server 8-bit East European	SB
WE8EBCDIC871	EBCDIC Code Page 871 8-bit Icelandic	SB
EL8EBCDIC875	EBCDIC Code Page 875 8-bit Greek	SB
EL8EBCDIC875R	EBCDIC Code Page 875 Server 8-bit Greek	SB
CL8EBCDIC1025	EBCDIC Code Page 1025 8-bit Cyrillic	SB
CL8EBCDIC1025C	EBCDIC Code Page 1025 Client 8-bit Cyrillic	SB
CL8EBCDIC1025R	EBCDIC Code Page 1025 Server 8-bit Cyrillic	SB
CL8EBCDIC1025S	EBCDIC Code Page 1025 Server 8-bit Cyrillic	SB
CL8EBCDIC1025X	EBCDIC Code Page 1025 (Modified) 8-bit Cyrillic	SB
BLT8EBCDIC1112	EBCDIC Code Page 1112 8-bit Baltic Multilingual	SB

Table A–5 European Language Character Sets (Cont.)

Name	Description	Comments
BLT8EBCDIC1112S	EBCDIC Code Page 1112 8-bit Server Baltic Multilingual	SB
D8EBCDIC1141	EBCDIC Code Page 1141 8-bit Austrian German	SB, EURO
DK8EBCDIC1142	EBCDIC Code Page 1142 8-bit Danish	SB, EURO
S8EBCDIC1143	EBCDIC Code Page 1143 8-bit Swedish	SB, EURO
I8EBCDIC1144	EBCDIC Code Page 1144 8-bit Italian	SB, EURO
F8EBCDIC1147	EBCDIC Code Page 1147 8-bit French	SB, EURO
EEC8EUROASCII	EEC Targon 35 ASCII West European/Greek	SB
EEC8EUROPA3	EEC EUROPA3 8-bit West European/Greek	SB
LA8PASSPORT	German Government Printer 8-bit All-European Latin	SB, ASCII
WE8HP	HP LaserJet 8-bit West European	SB
WE8ROMAN8	HP Roman8 8-bit West European	SB, ASCII
HU8CWI2	Hungarian 8-bit CWI-2	SB, ASCII
HU8ABMOD	Hungarian 8-bit Special AB Mod	SB, ASCII
LV8RST104090	IBM-PC Alternative Code Page 8-bit Latvian (Latin/Cyrillic)	SB, ASCII
US8PC437	IBM-PC Code Page 437 8-bit American	SB, ASCII
BG8PC437S	IBM-PC Code Page 437 8-bit (Bulgarian Modification)	SB, ASCII
EL8PC437S	IBM-PC Code Page 437 8-bit (Greek modification)	SB, ASCII
EL8PC737	IBM-PC Code Page 737 8-bit Greek/Latin	SB
LT8PC772	IBM-PC Code Page 772 8-bit Lithuanian (Latin/Cyrillic)	SB, ASCII
LT8PC774	IBM-PC Code Page 774 8-bit Lithuanian (Latin)	SB, ASCII
BLT8PC775	IBM-PC Code Page 775 8-bit Baltic	SB, ASCII
WE8PC850	IBM-PC Code Page 850 8-bit West European	SB, ASCII
EL8PC851	IBM-PC Code Page 851 8-bit Greek/Latin	SB, ASCII
EE8PC852	IBM-PC Code Page 852 8-bit East European	SB, ASCII
RU8PC855	IBM-PC Code Page 855 8-bit Latin/Cyrillic	SB, ASCII
WE8PC858	IBM-PC Code Page 858 8-bit West European	SB, ASCII, EURO
WE8PC860	IBM-PC Code Page 860 8-bit West European	SB, ASCII

Table A-5 European Language Character Sets (Cont.)

Name	Description	Comments
IS8PC861	IBM-PC Code Page 861 8-bit Icelandic	SB, ASCII
CDN8PC863	IBM-PC Code Page 863 8-bit Canadian French	SB, ASCII
N8PC865	IBM-PC Code Page 865 8-bit Norwegian	SB, ASCII
RU8PC866	IBM-PC Code Page 866 8-bit Latin/Cyrillic	SB, ASCII
EL8PC869	IBM-PC Code Page 869 8-bit Greek/Latin	SB, ASCII
LV8PC1117	IBM-PC Code Page 1117 8-bit Latvian	SB, ASCII
US8ICL	ICL EBCDIC 8-bit American	SB
WE8ICL	ICL EBCDIC 8-bit West European	SB
WE8ISOICLUK	ICL special version ISO8859-1	SB
WE8ISO8859P1	ISO 8859-1 West European	SB, ASCII
EE8ISO8859P2	ISO 8859-2 East European	SB, ASCII
SE8ISO8859P3	ISO 8859-3 South European	SB, ASCII
NEE8ISO8859P4	ISO 8859-4 North and North-East European	SB, ASCII
CL8ISO8859P5	ISO 8859-5 Latin/Cyrillic	SB, ASCII
AR8ISO8859P6	ISO 8859-6 Latin/Arabic	SB, ASCII
EL8ISO8859P7	ISO 8859-7 Latin/Greek	SB, ASCII, EURO
IW8ISO8859P8	ISO 8859-8 Latin/Hebrew	SB, ASCII
NE8ISO8859P10	ISO 8859-10 North European	SB, ASCII
BLT8ISO8859P13	ISO 8859-13 Baltic	SB, ASCII
CEL8ISO8859P14	ISO 8859-13 Celtic	SB, ASCII
WE8ISO8859P15	ISO 8859-15 West European	SB, ASCII, EURO
LA8ISO6937	ISO 6937 8-bit Coded Character Set for Text Communication	SB, ASCII
IW7IS960	Israeli Standard 960 7-bit Latin/Hebrew	SB
AR8ARABICMAC	Mac Client 8-bit Latin/Arabic	SB
EE8MACCE	Mac Client 8-bit Central European	SB
EE8MACCROATIAN	Mac Client 8-bit Croatian	SB
WE8MACROMAN8	Mac Client 8-bit Extended Roman8 West European	SB

Table A–5 European Language Character Sets (Cont.)

Name	Description	Comments
EL8MACGREEK	Mac Client 8-bit Greek	SB
IS8MACICELANDIC	Mac Client 8-bit Icelandic	SB
CL8MACCYRILLIC	Mac Client 8-bit Latin/Cyrillic	SB
AR8ARABICMACS	Mac Server 8-bit Latin/Arabic	SB, ASCII
EE8MACCES	Mac Server 8-bit Central European	SB, ASCII
EE8MACCROATIANS	Mac Server 8-bit Croatian	SB, ASCII
WE8MACROMAN8S	Mac Server 8-bit Extended Roman8 West European	SB, ASCII
CL8MACCYRILLICS	Mac Server 8-bit Latin/Cyrillic	SB, ASCII
EL8MACGREEKS	Mac Server 8-bit Greek	SB, ASCII
IS8MACICELANDICS	Mac Server 8-bit Icelandic	SB
BG8MSWIN	MS Windows 8-bit Bulgarian Cyrillic	SB, ASCII
LT8MSWIN921	MS Windows Code Page 921 8-bit Lithuanian	SB, ASCII
ET8MSWIN923	MS Windows Code Page 923 8-bit Estonian	SB, ASCII
EE8MSWIN1250	MS Windows Code Page 1250 8-bit East European	SB, ASCII, EURO
CL8MSWIN1251	MS Windows Code Page 1251 8-bit Latin/Cyrillic	SB, ASCII, EURO
WE8MSWIN1252	MS Windows Code Page 1252 8-bit West European	SB, ASCII, EURO
EL8MSWIN1253	MS Windows Code Page 1253 8-bit Latin/Greek	SB, ASCII, EURO
BLT8MSWIN1257	MS Windows Code Page 1257 8-bit Baltic	SB, ASCII, EURO
BLT8CP921	Latvian Standard LVS8-92(1) Windows/Unix 8-bit Baltic	SB, ASCII
LV8PC8LR	Latvian Version IBM-PC Code Page 866 8-bit Latin/Cyrillic	SB, ASCII
WE8NCR4970	NCR 4970 8-bit West European	SB, ASCII
WE8NEXTSTEP	NeXTSTEP PostScript 8-bit West European	SB, ASCII
CL8ISOIR111	ISOIR111 Cyrillic	SB
CL8KOI8R	RELCOM Internet Standard 8-bit Latin/Cyrillic	SB, ASCII
CL8KOI8U	KOI8 Ukrainian Cyrillic	SB
US8BS2000	Siemens 9750-62 EBCDIC 8-bit American	SB
DK8BS2000	Siemens 9750-62 EBCDIC 8-bit Danish	SB

Table A–5 European Language Character Sets (Cont.)

Name	Description	Comments
F8BS2000	Siemens 9750-62 EBCDIC 8-bit French	SB
D8BS2000	Siemens 9750-62 EBCDIC 8-bit German	SB
E8BS2000	Siemens 9750-62 EBCDIC 8-bit Spanish	SB
S8BS2000	Siemens 9750-62 EBCDIC 8-bit Swedish	SB
DK7SIEMENS9780X	Siemens 97801/97808 7-bit Danish	SB
F7SIEMENS9780X	Siemens 97801/97808 7-bit French	SB
D7SIEMENS9780X	Siemens 97801/97808 7-bit German	SB
I7SIEMENS9780X	Siemens 97801/97808 7-bit Italian	SB
N7SIEMENS9780X	Siemens 97801/97808 7-bit Norwegian	SB
E7SIEMENS9780X	Siemens 97801/97808 7-bit Spanish	SB
S7SIEMENS9780X	Siemens 97801/97808 7-bit Swedish	SB
EE8BS2000	Siemens EBCDIC.DF.04 8-bit East European	SB
WE8BS2000	Siemens EBCDIC.DF.04 8-bit West European	SB
WE8BS2000E	Siemens EBCDIC.DF.04 8-bit West European	SB, EURO
CL8BS2000	Siemens EBCDIC.EHC.LC 8-bit Cyrillic	SB
AL16UTF16	See "Universal Character Sets" on page A-18 for details	MB, EURO, FIXED
AL32UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTFE	See "Universal Character Sets" on page A-18 for details	MB, EURO

Middle Eastern Language Character Sets

[Table A–6](#) lists the Oracle character sets that can support Middle Eastern languages.

Table A–6 Middle Eastern Character Sets

Name	Description	Comments
AR8APTEC715	APTEC 715 Server 8-bit Latin/Arabic	SB, ASCII
AR8APTEC715T	APTEC 715 8-bit Latin/Arabic	SB
AR8ASMO708PLUS	ASMO 708 Plus 8-bit Latin/Arabic	SB, ASCII
AR8ASMO8X	ASMO Extended 708 8-bit Latin/Arabic	SB, ASCII
AR8ADOS710	Arabic MS-DOS 710 Server 8-bit Latin/Arabic	SB, ASCII
AR8ADOS710T	Arabic MS-DOS 710 8-bit Latin/Arabic	SB
AR8ADOS720	Arabic MS-DOS 720 Server 8-bit Latin/Arabic	SB, ASCII
AR8ADOS720T	Arabic MS-DOS 720 8-bit Latin/Arabic	SB
TR7DEC	DEC VT100 7-bit Turkish	SB
TR8DEC	DEC 8-bit Turkish	SB
WE8EBCDIC37C	EBCDIC Code Page 37 8-bit Oracle/c	SB
IW8EBCDIC424	EBCDIC Code Page 424 8-bit Latin/Hebrew	SB
IW8EBCDIC424S	EBCDIC Code Page 424 Server 8-bit Latin/Hebrew	SB
WE8EBCDIC500C	EBCDIC Code Page 500 8-bit Oracle/c	SB
IW8EBCDIC1086	EBCDIC Code Page 1086 8-bit Hebrew	SB
AR8EBCDIC420S	EBCDIC Code Page 420 Server 8-bit Latin/Arabic	SB
AR8EBCDICX	EBCDIC X BASIC Server 8-bit Latin/Arabic	SB
TR8EBCDIC1026	EBCDIC Code Page 1026 8-bit Turkish	SB
TR8EBCDIC1026S	EBCDIC Code Page 1026 Server 8-bit Turkish	SB
AR8HPARABIC8T	HP 8-bit Latin/Arabic	SB
TR8PC857	IBM-PC Code Page 857 8-bit Turkish	SB, ASCII
IW8PC1507	IBM-PC Code Page 1507/862 8-bit Latin/Hebrew	SB, ASCII
AR8ISO8859P6	ISO 8859-6 Latin/Arabic	SB, ASCII
IW8ISO8859P8	ISO 8859-8 Latin/Hebrew	SB, ASCII
WE8ISO8859P9	ISO 8859-9 West European & Turkish	SB, ASCII
LA8ISO6937	ISO 6937 8-bit Coded Character Set for Text Communication	SB, ASCII
IW7IS960	Israeli Standard 960 7-bit Latin/Hebrew	SB

Table A-6 Middle Eastern Character Sets (Cont.)

Name	Description	Comments
IW8MACHEBREW	Mac Client 8-bit Hebrew	SB
AR8ARABICMAC	Mac Client 8-bit Latin/Arabic	SB
AR8ARABICMACT	Mac 8-bit Latin/Arabic	SB
TR8MACTURKISH	Mac Client 8-bit Turkish	SB
IW8MACHEBREWS	Mac Server 8-bit Hebrew	SB, ASCII
AR8ARABICMACS	Mac Server 8-bit Latin/Arabic	SB, ASCII
TR8MACTURKISHS	Mac Server 8-bit Turkish	SB, ASCII
TR8MSWIN1254	MS Windows Code Page 1254 8-bit Turkish	SB, ASCII, EURO
IW8MSWIN1255	MS Windows Code Page 1255 8-bit Latin/Hebrew	SB, ASCII, EURO
AR8MSWIN1256	MS Windows Code Page 1256 8-Bit Latin/Arabic	SB, ASCII, EURO
IN8ISCII	Multiple-Script Indian Standard 8-bit Latin/Indian Languages	SB
AR8MUSSAD768	Mussa'd Alarabi/2 768 Server 8-bit Latin/Arabic	SB, ASCII
AR8MUSSAD768T	Mussa'd Alarabi/2 768 8-bit Latin/Arabic	SB
AR8NAFITHA711	Nafitha Enhanced 711 Server 8-bit Latin/Arabic	SB, ASCII
AR8NAFITHA711T	Nafitha Enhanced 711 8-bit Latin/Arabic	SB
AR8NAFITHA721	Nafitha International 721 Server 8-bit Latin/Arabic	SB, ASCII
AR8NAFITHA721T	Nafitha International 721 8-bit Latin/Arabic	SB
AR8SAKHR706	SAKHR 706 Server 8-bit Latin/Arabic	SB, ASCII
AR8SAKHR707	SAKHR 707 Server 8-bit Latin/Arabic	SB, ASCII
AR8SAKHR707T	SAKHR 707 8-bit Latin/Arabic	SB
AR8XBASIC	XBASIC 8-bit Latin/Arabic	SB
WE8BS2000L5	Siemens EBCDIC.DF.04.L5 8-bit West European/Turkish	SB
AL16UTF16	See "Universal Character Sets" on page A-18 for details	MB, EURO, FIXED
AL32UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTFE	See "Universal Character Sets" on page A-18 for details	MB, EURO

Universal Character Sets

[Table A-7](#) lists the Oracle character sets that provide universal language support, that is, they attempt to support all languages of the world, including, but not limited to, Asian, European, and Middle Eastern languages.

Table A-7 Universal Character Sets

Name	Description	Comments
AL16UTF16	Unicode 3.0 UTF-16 Universal character set	MB, EURO, FIXED
AL32UTF8	Unicode 3.0 UTF-8 Universal character set	MB, ASCII, EURO
UTF8	Unicode 3.0 UTF-8 Universal character set	MB, ASCII, EURO
UTFE	EBCDIC form of Unicode 3.0 UTF-8 Universal character set	MB, EURO

See Also: [Chapter 5, "Supporting Multilingual Databases with Unicode"](#)

Character Set Conversion Support

The following character set encodings are supported for conversion only, so they cannot be used as the database or national character set:

- AL16UTF16LE
- ISO2022-CN
- ISO2022-JP
- ISO2022-KR
- HZ-GB-2312

You can use these character sets as the `source_char_set` or `dest_char_set` in the `CONVERT` function.

See Also:

- *Oracle9i SQL Reference* for more information about the CONVERT function
- ["CONVERT Function"](#) on page 7-5

Subsets and Supersets

[Table A-8](#) lists common subset/superset relationships.

Table A-8 Subset-Superset Pairs

Subset	Superset
AR8ADOS710	AR8ADOS710T
AR8ADOS720	AR8ADOS720T
AR8ADOS720T	AR8ADOS720
AR8APTEC715	AR8APTEC715T
AR8ARABICMACT	AR8ARABICMAC
AR8ISO8859P6	AR8ASMO708PLUS
AR8ISO8859P6	AR8ASMO8X
AR8MUSSAD768	AR8MUSSAD768T
AR8MUSSAD768T	AR8MUSSAD768
AR8NAFITHA711	AR8NAFITHA711T
AR8NAFITHA721	AR8NAFITHA721T
AR8SAKHR707	AR8SAKHR707T
AR8SAKHR707T	AR8SAKHR707
BLT8CP921	BLT8ISO8859P13
D7DEC	D7SIEMENS9780X
D7SIEMENS9780X	D7DEC
DK7SIEMENS9780X	N7SIEMENS9780X
I7DEC	I7SIEMENS9780X
I7SIEMENS9780X	IW8EBCDIC424
IW8EBCDIC424	IW8EBCDIC1086
KO16KSC5601	KO16MSWIN949

Table A–8 Subset-Superset Pairs (Cont.)

Subset	Superset
LV8PC8LR	LV8RST104090
N7SIEMENS9780X	DK7SIEMENS9780X
US7ASCII	See Table 10–5 for a complete list
WE16DECTST	WE16DECTST2
WE16DECTST2	WE16DECTST
WE8DEC	TR8DEC
WE8DEC	WE8NCR4970
WE8ISO8859P1	WE8MSWIN1252
WE8NCR4970	TR8DEC
WE8NCR4970	WE8DEC
WE8PC850	WE8PC858

US7ASCII is a special case because so many other character sets are supersets of it. [Table A–9](#) lists supersets for US7ASCII.

Table A–9 US7ASCII Supersets

Supersets	Supersets	Supersets
AL24UTFFSS	EE8MACCES	NEE8ISO8859P4
AL32UTF8	EE8MACCROATIANS	RU8BESTA
AR8ADOS710	EE8MSWIN1250	RU8PC855
AR8ADOS710T	EE8PC852	RU8PC866
AR8ADOS720	EL8DEC	SE8ISO8859P3
AR8ADOS720T	EL8ISO8859P7	TH8MACTHAIS
AR8APTEC715	EL8MACGREEKS	TH8TISASCII
AR8APTEC715T	EL8MSWIN1253	TR8DEC
AR8ARABICMACS	EL8PC437S	TR8MACTURKISHS
AR8ASMO708PLUS	EL8PC851	TR8MSWIN1254
AR8ASMO8X	EL8PC869	TR8PC857
AR8HPARABIC8T	ET8MSWIN923	US8PC437

Table A-9 US7ASCII Supersets (Cont.)

Supersets	Supersets	Supersets
AR8ISO8859P6	HU8ABMOD	UTF8
AR8MSAWIN	HU8CWI2	VN8MSWIN1258
AR8MUSSAD768	IN8ISCII	VN8VN3
AR8MUSSAD768T	IS8PC861	WE8DEC
AR8NAFITHA711	IW8ISO8859P8	WE8DG
AR8NAFITHA711T	IW8MACHEBREWS	WE8ISO8859P1
AR8NAFITHA721	IW8MSWIN1255	WE8ISO8859P15
AR8NAFITHA721T	IW8PC1507	WE8ISO8859P9
AR8SAKHR706	JA16EUC	WE8MACROMAN8S
AR8SAKHR707	JA16SJIS	WE8MSWIN1252
AR8SAKHR707T	JA16TSTSET	WE8NCR4970
BG8MSWIN	JA16TSTSET2	WE8NEXTSTEP
BG8PC437S	JA16VMS	WE8PC850
BLT8CP921	KO16KSC5601	WE8PC858
BLT8ISO8859P13	KO16KSCCS	WE8PC860
BLT8MSWIN1257	KO16MSWIN949	WE8ROMAN8
BLT8PC775	KO16TSTSET	ZHS16CGB231280
BN8BSCII	LA8ISO6937	ZHS16GBK
CDN8PC863	LA8PASSPORT	ZHT16BIG5
CEL8ISO8859P14	LT8MSWIN921	ZHT16CCDC
CL8ISO8859P5	LT8PC772	ZHT16DBT
CL8KOI8R	LT8PC774	ZHT16HKSCS
CL8KOI8U	LV8PC1117	ZHT16MSWIN950
CL8ISOIR111	LV8PC8LR	ZHT32EUC
CL8MACCYRILLICS	LV8RST104090	ZHT32SOPS
CL8MSWIN1251	N8PC865	ZHT32TRIS
EE8ISO8859P2	NE8ISO8859P10	

Linguistic Sorting

Oracle offers two kinds of linguistic sorts, monolingual and multilingual. In addition, monolingual sorts can be extended to handle special cases. These special cases (represented with a prefix X) typically mean that the characters will be sorted differently from their ASCII values. For example, *ch* and *ll* are treated as only one character in XSPANISH. In other words, the SPANISH sort is uses modern collation rules while XSPANISH uses traditional sorting rules.

[Table A–10](#) lists the monolingual linguistic sorts supported by the Oracle server.

Table A–10 *Monolingual Linguistic Sorts*

Basic Name	Extended Name	Special Cases
ARABIC	--	
ARABIC_MATCH	--	
ARABIC_ABI_SORT	--	
ARABIC_ABI_MATCH	--	
ASCII7	--	
BENGALI	--	
BIG5		
BINARY		
BULGARIAN	--	
CANADIAN FRENCH	--	
CATALAN	XCATALAN	æ, AE, ß
CROATIAN	XCROATIAN	D, L, N, d, l, n, ß
CZECH	XCZECH	ch, CH, Ch, ß
CZECH_PUNCTUTION	XCZECH_PUNCTUATION	ch, CH, Ch, ß
DANISH	XDANISH	A, ß, Å, à
DUTCH	XDUTCH	ij, IJ
EBCDIC		
EEC_EURO	--	
EEC_EUROPA3	--	

Table A–10 Monolingual Linguistic Sorts (Cont.)

Basic Name	Extended Name	Special Cases
ESTONIAN	--	
FINNISH	--	
FRENCH	XFRENCH	
GERMAN	XGERMAN	ß
GERMAN_DIN	XGERMAN_DIN	ß, ä, ö, ü, Ä, Ö, Ü
GBK		
GREEK	--	
HEBREW	--	
HKSCS		
HUNGARIAN	XHUNGARIAN	cs, gy, ny, sz, ty, zs, ß, CS, Cs, GY, Gy, NY, Ny, SZ, Sz, TY, Ty, ZS, Zs
ICELANDIC	--	
INDONESIAN	--	
ITALIAN	--	
JAPANESE	--	
LATIN	--	
LATVIAN	--	
LITHUANIAN	--	
MALAY	--	
NORWEGIAN	--	
POLISH	--	
PUNCTUATION	XPUNCTUATION	
ROMANIAN	--	
RUSSIAN	--	
SLOVAK	XSLOVAK	dz, DZ, Dz, ß (<i>caron</i>)
SLOVENIAN	XSLOVENIAN	ß
SPANISH	XSPANISH	ch, ll, CH, Ch, LL, Ll

Table A–10 Monolingual Linguistic Sorts (Cont.)

Basic Name	Extended Name	Special Cases
SWEDISH	--	
SWISS	XSWISS	ß
THAI_DICTIONARY	--	
THAI_TELEPHONE	--	
TURKISH	XTURKISH	æ, AE, ß
UKRAINIAN	--	
UNICODE_BINARY		
VIETNAMESE	--	
WEST_EUROPEAN	XWEST_EUROPEAN	ß

Table A–11 lists the multilingual linguistic sorts available in Oracle. All of them include `GENERIC_M` (an ISO standard for sorting Latin-based characters) as a base. Multilingual linguistic sorts are for the collation of a given primary language together with Latin-based characters. For example, `KOREAN_M` will sort Korean and Latin-based characters, but it will not collate Chinese, Thai, or Japanese characters.

Table A–11 *Multilingual Linguistic Sorts*

Basic Name	Explanation
CANADIAN_M	Canadian French sort supports reverse secondary, special expanding characters
DANISH_M	Danish sort supports sorting lower case characters before upper case characters
FRENCH_M	French sort supports reverse sort for secondary
GENERIC_M	Generic sorting order which is based on ISO14651 and Unicode canonical equivalence rules but excluding compatible equivalence rules
JAPANESE_M	Japanese sort supports SJIS character set order and EUC characters which are not included in SJIS
KOREAN_M	Korean sort: Hangul characters are based on Unicode binary order. Hanja characters based on pronunciation order. All Hangul characters are before Hanja characters
SPANISH_M	Traditional Spanish sort supports special contracting characters
THAI_M	Thai sort supports swap characters for some vowels and consonants
SCHINESE_RADICAL_M	Simplified Chinese sort based on radical as primary order and number of strokes order as secondary order
SCHINESE_STROKE_M	Simplified Chinese sort uses number of strokes as primary order and radical as secondary order
SCHINESE_PINYIN_M	Simplified Chinese PinYin sorting order
TCHINESE_RADICAL_M	Traditional Chinese sort based on radical as primary order and number of strokes order as secondary order
TCHINESE_STROKE_M	Traditional Chinese sort uses number of strokes as primary order and radical as secondary order

Calendar Systems

By default, most territory definitions use the Gregorian calendar system. [Table A-12](#) lists the other calendar systems supported by the Oracle server.

Table A-12 *NLS Supported Calendars*

Name	Default Format	Character Set Used For Default Format
Japanese Imperial	EEYYMMDD	JA16EUC
ROC Official	EEyymmdd	ZHT32EUC
Thai Buddha	dd month EE yyyy	TH8TISASCII
Persian	DD Month YYYY	AR8ASMO8X
Arabic Hijrah	DD Month YYYY	AR8ISO8859P6
English Hijrah	DD Month YYYY	AR8ISO8859P6

[Figure A-1](#) shows how March 20, 1998 appears in ROC Official:

Figure A-1 ROC Official Example

```
SQL> alter session set NLS_CALENDAR='ROC Official';
Session altered.

SQL> alter session set NLS_DATE_FORMAT =
2 ' "中華民國"YY"年"MM"月"DD"日";
Session altered.

SQL> select sysdate from dual;

SYSDATE
-----
中華民國87年03月20日
```

[Figure A-2](#) shows how March 27, 1998 appears in Japanese Imperial:

Figure A–2 Japanese Imperial Example

```
SQL> alter session set NLS CALENDAR =
      2 'Japanese Imperial';

Session altered.

SQL> alter session set NLS DATE FORMAT=
      2 '"平成"YY"年"MM"月"DD"日"'

Session altered.

SQL> select sysdate from dual;

SYSDATE
-----
平成10年03月27日
```

Obsolete Locale Data

Before Oracle server release 7.2, when a character set was renamed, the old name was usually supported along with the new name for several releases after the change. Beginning with release 7.2, the old names are no longer supported.

[Table A–13](#) lists the affected character sets. If you reference any of these character sets in your code, replace them with their new name:

Table A–13 New Names for Obsolete NLS Data Character Sets

Old Name	New Name
AL24UTFSS	UTF8, AL32UTF8
AR8MSAWIN	AR8MSWIN1256

Table A–13 New Names for Obsolete NLS Data Character Sets (Cont.)

Old Name	New Name
CL8EBCDIC875S	CL8EBCDIC875R
EL8EBCDIC875S	EL8EBCDIC875R
JVMS	JA16VMS
JEUC	JA16EUC
SJIS	JA16SJIS
JDBCS	JA16DBCS
KSC5601	KO16KSC5601
KDBCS	KO16DBCS
CGB2312-80	ZHS16CGB231280
CNS 11643-86	ZHT32EUC
JA16EUCFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHS32EUCFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHS16GBKFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
JA16DBCSFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
KO16DBCSFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHS16DBCSFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHS16CGB231280 FIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHT16DBCSFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
KO16KSC5601FIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
JA16SJISFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHT16BIG5FIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.

Table A–13 *New Names for Obsolete NLS Data Character Sets (Cont.)*

Old Name	New Name
ZHT32TRISFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.

Character set CL8MSWINDOW31 has been desupported. The newer character set CL8MSWIN1251 is actually a duplicate of CL8MSWINDOW31 and includes some characters omitted from the earlier version. Change any usage of CL8MSWINDOW31 to CL8MSWIN1251 instead.

AL24UTFFSS Character Set Desupport

The Unicode Character Set AL24UTFFSS has been desupported in Oracle9i. AL24UTFFSS was introduced with Oracle7 as the Unicode character set supporting UTF-8 encoding scheme based on the Unicode standard 1.1, which is now obsolete. In Oracle9i, Oracle now offers the Unicode database character set AL32UTF8 and UTF8, which includes the Unicode enhancements based on the Unicode standard 3.0.

The migration path for an existing AL24UTFFSS database is to upgrade to UTF8 prior to upgrading to Oracle9i. As with all migrations to a new database character set, Oracle Corporation recommends that you use the Character Set Scanner for data analysis before attempting to migrate your existing database character set to UTF8.

See Also: [Chapter 10, "Character Set Scanner Utility"](#)

Unicode Character Code Assignments

This appendix offers an introduction to how Unicode assigns characters. This appendix contains:

- [Unicode Character Code Assignments](#)

Unicode Character Code Assignments

Table B–1 contains Unicode details.

Table B–1 Unicode Character Code Assignments

Characters	UTF-16 Character Codes		UTF-8 Character Codes			
	First 16-bits	Second 16-bits	First Byte	Second Byte	Third Byte	Fourth Byte
ASCII	000-007F		00-7F			
European (except ASCII), Arabic, Hebrew, etc.	0080-07FF		C2-DF	80-BF		
Indic, Thai, certain symbols (for example, euro), Chinese, Japanese, Korean, etc.	0800-0FFF		E0	A0-BF	80-BF	
	1000 - CFFF		E1-EC	80-BF	80-BF	
	D000 - D7FF		ED	80-9F	80-0BF	
	F900-FFFF		EF	A4-BF	80-BF	
Private Use Area #1	E000 - EFFF		EE	80-BF	80-BF	
	F000 - F8FF		EF	80-A3	80-BF	
Additional Chinese/Japanese/Korean characters, historic characters, musical and mathematical symbols, etc.	D800 - D8BF	DC00 - DFFF	F0	90-BF	80-BF	80-BF
	D8C0 - DABF	DC00 - DFFF	F1-F2	80-BF	80-BF	80-BF
	DAC0 - DB7F	DC00 - DFFF	F3	80-AF	80-BF	80-BF
Private Use Area #2	DB80 - DBBF	DC00 - DFFF	F3	B0-BF	80-BF	80-BF
	DBC0 - DBFF	DC00 - DFFF	F4	80-8F	80-BF	80-BF

Note: Blank spaces represent non-applicable code assignments. Character codes in this table are shown in hexadecimal representation

UTF-16 Encoding

As shown in [Table B-1](#), UTF-16 character codes for some characters (Additional Chinese/Japanese/Korean characters and Private Use Area #2) are represented in two units of 16-bits. These are the surrogate pairs. A surrogate pair consists of two 16-bit values. The first 16-bit value is the high surrogate (the values are from 0xD800 to 0xDBFF). The second 16-bit value is the low surrogate (the values are from 0xDC00 to 0xDFFF). With surrogate pairs, UTF-16 character codes can represent more than one million characters. Without surrogate pairs, only up to 65,536 characters could be represented. Oracle's AL16UTF16 character set supports surrogate pairs.

See Also: ["Surrogate Characters"](#) on page 4-7 for further information regarding surrogate pairs

UTF-8 Encoding

The UTF-8 character codes in [Table B-1](#) show that:

- ASCII characters use 1-byte
- European (except ASCII), Arabic, and Hebrew characters require 2-bytes
- Indic, Thai, Chinese, Japanese, and Korean characters as well as certain symbols such as the one for the euro require 3-bytes
- Characters in the Private Use Area #1 require 3-bytes
- Additional Chinese/Japanese/Korean etc. characters require 4-bytes
- Characters in the Private Use Area #2 require 4-bytes

Oracle's AL32UTF8 character set supports 1-byte, 2-byte, 3-byte, and 4-byte values. Oracle's UTF8 character set supports 1-byte, 2-byte, and 3-byte values, but not 4-byte values.

Glossary

AL16UTF16

The Unicode UTF-16 national character set.

AL32UTF8

The Unicode 3.0 UTF-8 database character set with 4-byte surrogate pairs support.

ASCII

American Standard Code for Information Interchange. A common encoded 7-bit character set for English. ASCII includes the letters A-Z and a-z, as well as digits, punctuation symbols, and control characters. The Oracle character set name for this is US7ASCII.

binary sorting

Sorting of character strings based on their binary coded value representations.

byte semantics

Byte semantics means treating strings as a sequence of bytes.

case conversion

Case conversion refers to changing a character from its uppercase to lowercase form, or vice versa.

character

A character is an abstract element of a text. A character is different from a glyph (font glyph), which is a specific instance of a character. For example, the first character of the English upper-case Alphabet can be printed (or displayed) as A, Å,

A, etc. All these different forms are different glyphs but represent the same character. A character, a character code and a glyph are related as follows.

character --(encoding)--> character code --(font)--> glyph

For example, the first character of the English upper-case alphabet is represented in computer memory as a number (or a character code). The character code is 0x41 if we are using the ASCII encoding scheme, or the character code is 0xc1 if we are using the EBCDIC encoding scheme, or it can be some other number if we are using different encoding scheme. When we print or display this character, we use a font. We have to choose a font for the ASCII encoding scheme (or a font for a superset of the ASCII encoding scheme) if we are using the ASCII encoding scheme, or we have to choose a font for the EBCDIC encoding scheme if we are using the EBCDIC encoding scheme. Now the character is printed (or displayed) as A, A, A, or some other form. All these different forms are different glyphs, but represent the same character.

character code

A character code is a number which represents a specific character. In order for computers to handle a character, we need a specific number which is assigned to that character. The number (or the character code) depends on what encoding scheme we are using. For example, the first character of the English uppercase alphabet has the character code 0x41 for the ASCII encoding scheme, but the same character has the character code 0xc1 for the EBCDIC encoding scheme.

See also **character**.

character semantics

Character semantics means treating strings as a sequence of characters, as opposed to bytes semantics, where strings are counted in bytes.

character set

A character set is a set of characters for a specific language or group of languages. There can be many different character sets just for one language.

A character set does not always imply any specific character encoding scheme.

In this manual, a character set generally implies a specific character encoding scheme, which is how a character code is assigned to each character of the character set. Therefore, the meaning of the term character set is generally the same as encoded character set in this manual.

character string

A character string is a serial string of characters or even no character. In this case, the character string is called a "null string". "The number of characters" of this character string is 0 (zero).

coded character set

Same as encoded character set.

An independent unit used to represent data, such as a letter, a letter with a diacritical mark, a digit, ideograph, punctuation, or symbol.

character classification

Character classification information provides details about the type of character associated with each legal character code; that is, whether it is an alphabetic, uppercase, lowercase, punctuation, control, or space character, etc.

character encoding scheme

A character encoding scheme is a rule that assigns numbers (or character codes) to all characters in a character set. We also use the shortened term encoding scheme (or encoding method, or just encoding).

client character set

The encoded character set which the client uses. A client character set can differ from the database server character set, in which case, character set conversion must occur.

collation

Ordering of character strings in a given alphabet in a linguistic sort order or a binary sort order.

combining character

A character that graphically combines with a preceding base character. These characters are not used in isolation. They include such characters as accents, diacritics, Hebrew points, Arabic vowel signs, and Indic matras.

composite character

A single character which can be represented by a composite character sequence. This type of character is found in the scripts of Thai, Lao, Vietnamese, and Korean Hangul, as well as many Latin characters used in European languages.

composite character sequence

A character sequence consisting of a base character followed by one or more combining characters. This is also referred to as a combining character sequence.

database character set

The encoded character set in which text is stored in the database is represented. This includes CHAR, VARCHAR2, LONG, and fixed-width CLOB column values and all SQL and PL/SQL text stored in the database.

diacritical mark

A mark added to a letter that usually provides information about pronunciation or stress. The letter "ä" is an example of a diacritical mark added to the letter "a".

EBCDIC

Extended Binary Coded Decimal Interchange Code. EBCDIC is a family of encoded character sets used mostly on IBM systems.

encoded character set

An encoded character set is a character set with an associated character encoding scheme. An encoded character set specifies how a number (or a character code) is assigned to each character of the character set based on a character encoding scheme.

encoding

Encoding Method or Encoding scheme. *See also* **character encoding scheme**.

font

An ordered collection of character glyphs which provides a graphical representation of characters within a character set.

globalization

The process of making software flexible enough to be used in many different linguistic and cultural environments. Globalization should not be confused with localization, which is the process of preparing software for use in one specific locale.

glyph

A glyph (font glyph) is a specific instance of a character. A character can have many different glyphs. For example, the first character of the English upper-case Alphabet can be printed (or displayed) as A, Å, Ä, etc.

All these different forms are different glyphs, but representing the same character.
See also character.

ideograph

A symbol representing an idea. Chinese is an example of an ideographic writing system.

internationalization

The process of making software flexible enough to be used in many different linguistic and cultural environments. Internationalization should not be confused with localization, which is the process of preparing software for use in one specific locale.

ISO

International Organization for Standards. A worldwide federation of national standards bodies from 130 countries. The mission of ISO is to promote the development of standardization and related activities in the world with a view to facilitating the international exchange of goods and services.

ISO 14651

A multilingual sort designed to handle almost all languages of the world.

ISO/IEC 10646

A universal character set standard defining the characters of most major scripts used in the modern world. In 1993, ISO adopted Unicode version 1.1 as ISO/IEC 10646-1:1993. ISO/IEC 10646 has two formats: UCS-2 is a 2-byte fixed-width format and UCS-4 is a 4-byte fixed-width format. There are three levels of implementation, all relating to support for composite characters. Level 1 requires no composite character support, level 2 requires support for specific scripts (including most of the Unicode scripts such as Arabic, Thai, etc.), and level 3 requires unrestricted support for composite characters in all languages.

ISO currency

The 3-letter abbreviation used to denote a local currency, which is based on the ISO 4217 standard. For example, "USD" represents the United States Dollar.

ISO 8859

A family of 8-bit encoded character sets. The most common one is ISO 8859-1 (also known as Latin-1), and is used for Western European languages.

ISO 14651

An International String Ordering standard sort designed to handle almost all languages.

Latin-1

Formally known as the ISO 8859-1 character set standard. An 8-bit extension to ASCII which adds 128 characters covering the most common Latin characters used in Western Europe. The Oracle character set name for this is WE8ISO8859P1. See "ISO 8859".

length semantics

Length semantics determines how you treat stringlengths. They can be treated as a sequence of characters or bytes.

linguistic index

An index built on a linguistic collation order.

linguistic sort

A sort of strings based on requirements from a locale instead of based on the binary representation of the strings. *See also* **multilingual linguistic sort** and **monolingual linguistic sort**.

locale

A collection of information regarding the linguistic and cultural preferences from a particular region. Typically, a locale consists of language, territory, character set, linguistic, and calendar information defined in NLS data files.

Locale Builder

A GUI utility that offers a way to modify, view or define locale-specific data. You can also create your own formats for language, territory, character set, and collation.

localization

The process of providing language-specific or culture-specific information for software systems. Translation of an application's user interface would be an example of localization. Localization should not be confused with internationalization, which is the process of generalizing software so it can handle many different linguistic and cultural conventions.

monolingual linguistic sort

An Oracle sort that uses two passes when comparing strings. This is fine for most European languages, but is inadequate for Asian languages. *See also multilingual linguistic sort.*

monolingual support

Support for only one language.

multibyte

Multibyte means characters represented by two or more bytes.

When character codes are assigned to all characters in a specific language (or a group of languages), one byte (8 bits) can represent 256 different characters. Two bytes (16 bits) can represent up to 65,536 different characters. However, two bytes are still not enough to represent all the characters for many languages. We use 3 bytes or 4 bytes for those characters.

One example is the UTF8 encoding of Unicode. In UTF8, there are many 2-byte and 3-byte characters.

Another example is Traditional Chinese language used in Taiwan. It has more than 80,000 different characters. Some character encoding schemes used in Taiwan encode characters in up to 4 bytes.

multibyte character

A multibyte character is a character whose character code consists of two or more bytes under a certain character encoding scheme. Note that the same character may have different character code where the character encoding scheme is different. Without knowing which character encoding scheme is being used, Oracle cannot tell which character is a multibyte character. For example, Japanese Hankaku-Katakana (half width Katakana) characters are one byte in JA16SJIS encoded character set, two bytes in JA16EUC, and three bytes in UTF8. See "single-byte character".

multibyte character string

A multibyte character string is a character string which consists of one of the below.

- No characters
(The character string is called the "null string" in this case.)
- One or more single-byte character(s)

- A mixture of one or more single-byte characters and one or more multibyte characters
- One or more multibyte characters

multilingual linguistic sort

An Oracle sort that uses evaluates strings on three levels when comparing.

National character set

An alternate character set from the database character set that can be specified for NCHAR, NVARCHAR2, and NCLOB columns. National character sets are in Unicode only.

NLB files

Binary files used by the Locale Builder to define locale-specific data.

NLS

National Language Support. NLS allows users to interact with the database in their native languages. It also allows applications to run in different linguistic and cultural environments. The term is somewhat obsolete because Oracle supports global users at one time.

NLSDATA

A general phrase referring to the contents in many files with .nlb suffixes. These files contain data that the NLSRTL library uses to provide specific NLS support.

NLSRTL

National Language Support Run-Time Library. This library is responsible for providing locale-independent algorithms for internationalization. The locale-specific information (that is, NLSDATA) is read by the NLSRTL library during run-time.

NLT files

Text files used by the Locale Builder to define locale-specific data. Because they are in text, you can view the settings.

replacement character

A character used during character conversion when the source character is not available in the target character set. For example, ? is often used as Oracle's default replacement character.

restricted multilingual support

Multilingual support which is restricted to a group of related languages. Support for related languages, but not all languages. Similar language families, such as Western European languages can be represented with, for example, ISO 8859/1. In this case, however, Thai could not be added.

SQL CHAR datatypes

Includes CHAR, VARCHAR, VARCHAR2, CLOB, and LONG datatypes.

SQL NCHAR datatypes

Includes NCHAR, NVARCHAR, NVARCHAR2, and NCLOB datatypes.

script

A collection of related graphic symbols used in a writing system. Some scripts are used to represent multiple languages, and some languages use multiple scripts. Example of scripts include Latin, Arabic, and Han.

single-byte

Single-byte (or single byte) means one byte. One byte usually consists of 8 bits. When we assign character codes to all characters for a specific language, one byte (8 bits) can represent 256 different characters.

single-byte character

A single-byte character is a character whose character code consists of one byte under a certain character encoding scheme. Note that the same character may have different character code where the character encoding scheme is different. Without knowing which character encoding scheme we are using, we cannot tell which character is a single-byte character. For example, the euro currency symbol is one byte in WE8MSWIN1252 encoded character set, two bytes in AL16UTF16, and three bytes in UTF8. *See also* **multibyte character**.

single-byte character string

A single-byte character string is a character string that consists of one of the below.

- No character
(The character string is called "null string" in this case.)
- One or more single-byte characters.

surrogate characters

You can extend Unicode to encode more than 1 million characters. These extended characters are called surrogate pairs. Surrogate pairs are designed to allow representation of characters in future extensions of the Unicode standard. Surrogate pairs require 4 bytes in UTF-8 and UTF-16,

UCS-2

UCS stands for "Universal Multiple-Octet Coded Character Set". It is a 1993 ISO and IEC standard character set. Fixed-width 16-bit Unicode. Each character occupies 16 bits of storage. The Latin-1 characters are the first 256 code points in this standard, so it can be viewed as a 16-bit extension of Latin-1.

UCS-4

Fixed-width 32-bit Unicode. Each character occupies 32 bits of storage. The UCS-2 characters are the first 65,536 code points in this standard, so it can be viewed as a 32-bit extension of UCS-2. This is also sometimes referred to as ISO-10646. ISO-10646 is a standard that specifies up to 2,147,483,648 characters in 32768 planes, of which the first plane is the UCS-2 set. The ISO standard also specifies transformations between different encodings.

Unicode

Unicode is a type of universal character set, a collection of 64K characters encoded in a 16-bit space. It encodes nearly every character in just about every existing character set standard, covering most written scripts used in the world. It is owned and defined by Unicode Inc. Unicode is canonical encoding which means its value can be passed around in different locales. But it does not guarantee a round-trip conversion between it and every Oracle character set without information loss.

Unicode codepoint

A 16-bit binary value that can represent a unit of encoded text for processing and interchange. Every point between U+0000 and U+FFFF is a code point. The term Unicode codepoint is interchangeable with code element, code position, and code value.

Unicode datatype

A SQL NCHAR datatype (NCHAR, NVARCHAR2, and NCLOB). You can store Unicode characters into columns of these datatypes irrespective of the database character set.

unrestricted multilingual support

Being able to use as many languages as desired. A universal character set, such as Unicode, helps to provide unrestricted multilingual support because it supports a very large character repertoire, encompassing most modern languages of the world.

UTFE

The Unicode 3.1 UTF-8 database character set with 6 byte surrogate pairs support.

UTF-8

A variable-width encoding of UCS-2 that uses sequences of 1, 2, or 3 bytes per character. Characters from 0-127 (the 7-bit ASCII characters) are encoded with one byte, characters from 128-2047 require two bytes, and characters from 2048-65535 require three bytes. The Oracle character set name for this is UTF8. The standard has left room for expansion to support the UCS4 characters with sequences of 4, 5, and 6 bytes per character.

UTF-16

An extension to UCS-2 that allows for pairs of UCS-2 code points to represent extended characters from the UCS-4 set. UCS-2 has ranges of code points allocated for high (leading) and low (trailing) surrogates that support UTF-16 encodings.

wide character

A fixed-width character format that is well-suited for extensive text processing because it allows for data to be processed in consistent fixed-width chunks. Wide characters are intended for supporting internal character processing, and are therefore implementation-dependent.

Index

A

- abbreviations
 - AM/PM, 3-17
 - BC/AD, 3-17
 - languages, A-2
- AL32UTF8 character set, 5-5
- ALTER DATABASE CHARACTER SET
 - statement, 10-7
- ALTER DATABASE NATIONAL CHARACTER SET statement, 10-7
- ALTER SESSION statement
 - SET NLS_CURRENCY clause, 3-27, 3-28
 - SET NLS_DATE_FORMAT clause, 3-15
 - SET NLS_LANGUAGE clause, 3-12
 - SET NLS_NUMERIC_CHARACTERS clause, 3-26
 - SET NLS_TERRITORY clause, 3-12
- ALTER SYSTEM statement
 - SET NLS_LANGUAGE clause, 3-12
- AM/PM abbreviation
 - language of, 3-17
- array parameter
 - Character Set Scanner Utility, 10-16
- ASCII encoding, 2-6

B

- BC/AD abbreviation
 - language of, 3-17
- BLANK_TRIMMING paramter, 10-4
- boundaries parameter
 - Character Set Scanner Utility, 10-17

C

- calendar systems
 - support, A-26
- calendars, A-26
 - customized, 12-7
 - formats, 3-21
 - parameter, 3-21
 - systems, 3-23
- capture parameter
 - Character Set Scanner Utility, 10-17
- CHAR
 - class, 9-8
- character set
 - conversion, 2-11
 - encoding, 2-2
 - setting definition, 11-16
- character set migration
 - data scanning, 10-7
- Character Set Scanner
 - scan modes, 10-11
- Character Set Scanner Utility, 10-1, 10-9, 10-18
 - array parameter, 10-16
 - boundaries parameter, 10-17
 - capture parameter, 10-17
 - compatibility, 10-13
 - feedback parameter, 10-18
 - fromnchar parameter, 10-18
 - full parameter, 10-19
 - help parameter, 10-19
 - invoking, 10-14
 - lastrpt parameter, 10-19, 10-20
 - maxblocks parameter, 10-20
 - online help, 10-14

- parameter file, 10-15
- parameters, 10-16
- scanner messages, 10-40
- scanner parameters, 10-16
- scanner tables, 10-37
- suppress parameter, 10-21
- table parameter, 10-21
- tochar parameter, 10-22
- user parameter, 10-22
- userid parameter, 10-23
- character sets
 - 8-bit versus 7-bit, 7-5
 - Asian, A-8
 - choosing, 10-2
 - conversion, 2-16
 - conversion using OCI, 8-35
 - converting, 7-5
 - data loss, 10-4
 - European, A-9
 - Middle Eastern, A-15
 - migrating, 10-2
 - migration, 10-2
 - naming, 2-10
 - parameters, 3-34
 - restrictions on expressing names and text, 2-13
 - storage, A-6
 - supported, 2-15
 - universal, A-18
- choosing character sets, 10-2
- collation
 - using Locale Builder Utility, 11-25
- collation parameters, 3-31
- concatenation operator, 7-14
- conversion
 - between character set ID number and character set name, 7-9
- CONVERT function, 7-5
- converting
 - character sets, 7-5
- CSM\$COLUMNS parameter, 10-35
- CSM\$ERRORS parameter, 10-36
- CSM\$TABLES parameter, 10-35
- CSMIG user, 10-13
- CSMINST.SQL script, 10-13
 - running, 10-13

- currencies
 - formats, 3-26
 - monetary
 - units characters, 3-30
 - symbols
 - default, 3-10
 - local currency symbol, 3-27
- customized
 - calendars, 12-7
 - character sets, 12-2

D

- data
 - conversion, 7-5
- data conversion
 - database character set, 10-7
- data expansion during character set migration, 10-2
- data inconsistencies
 - causing data loss, 10-5
- data loss
 - caused by data inconsistencies, 10-5
 - during character set migration, 10-4
 - from mixed character sets, 10-6
- data scanning
 - character set migration, 10-7
- data truncation, 10-2
 - restrictions, 10-3
- database character set
 - choosing, 2-10
 - conversion, 10-7
- database character set migration, 10-6
- Database Scan Summary Report, 10-26, 10-27
- date formats, 3-14, 7-12
 - and partition bound expressions, 3-16
- dates
 - ISO standard, 3-22, 7-13
 - NLS_DATE_LANGUAGE parameter, 3-16
- days
 - format element, 3-17
 - language of names, 3-17
- decimal character
 - default, 3-10
 - NLS_NUMERIC_CHARACTERS

- parameter, 3-25
 - when not a period (.), 3-25
- drivers
 - JDBC, 9-2

E

- EJB, 9-25
- encoding schemes, 2-8
- Enterprise Java Beans, 9-25
- explicit authentication, 9-25

F

- feedback parameter
 - Character Set Scanner Utility, 10-18
- format elements, 7-12, 7-13
 - C, 7-14
 - D, 3-25, 7-13
 - day, 3-17
 - G, 3-25, 7-13
 - IW, 7-13
 - IY, 7-13
 - L, 3-27, 7-13
 - month, 3-17
 - RM, 3-15, 7-12
 - RN, 7-14
- formats
 - calendar, 3-21
 - currency, 3-26
 - numeric, 3-24
- fromchar parameter, 10-18
 - Character Set Scanner Utility, 10-18
- fromnchar parameter
 - Character Set Scanner Utility, 10-18
- full parameter
 - Character Set Scanner Utility, 10-19

G

- getString() method, 9-8
- getStringWithReplacement() method, 9-8
- Globalization
 - architecture, 1-2
- globalization features, 1-5

- group separator, 3-25
 - default, 3-10
- NLS_NUMERIC_CHARACTERS
 - parameter, 3-25

H

- help parameter
 - Character Set Scanner Utility, 10-19

I

- implicit authentication, 9-25
- indexes
 - partitioned, 7-12
- Individual Exception Report, 10-26, 10-33
- ISO standard
 - date format, 3-22, 7-13
- ISO week number, 7-13
- IW format element, 7-13
- IY format element, 7-13

J

- Java runtime environment, 9-3
- Java stored procedures, 9-16
- Java Virtual Machine, 9-14
- java.sql.ResultSet, 9-4
- JDBC
 - class library, 9-6
 - drivers, 9-2
 - OCI driver
 - NLS considerations, 9-6
 - Server driver, 9-7
 - Thin driver
 - NLS considerations, 9-7
- JDBC drivers
 - and NLS, 9-4
- JVM, 9-14

L

- L format element, 3-27
- language definition
 - setting, 11-8
- language support, 1-6

- languages
 - overriding, 3-6
- lastprt parameter
 - Character Set Scanner Utility, 10-19, 10-20
- linguistic definitions, A-22
 - supported, A-22
- linguistic sorts
 - controlling, 7-12
- list separator, 3-34
- local currency symbol, 3-27
- Locale Builder Utility, 11-3
 - restrictions, 11-5
- LXEGEN executable, 12-8
- LXINST executable, 12-8

M

- maxblocks parameter
 - Character Set Scanner Utility, 10-20
- messages
 - error, A-4
 - translated, A-4
- migrating character sets, 10-2
- migration
 - database character set, 10-6
- mixed character sets
 - causing data loss, 10-6
- monetary
 - parameters, 3-26
 - units characters, 3-30
- months
 - format element, 3-17
 - language of names, 3-17

N

- National Language Support (NLS)
 - NLS_LANGUAGE parameter, 7-5
- NCHAR datatype migration, 5-16
- NLB files, 11-2
- NLS
 - and JDBC drivers, 9-4
 - conversions, 9-4
 - for JDBC OCI drivers, 9-6
 - for JDBC Thin drivers, 9-7

- Java methods that employ, 9-4
- NLS Calendar Utility, 12-7
- NLS data
 - error messages, A-4
 - supported calendar systems, A-26
 - supported linguistic definitions, A-22
 - supported territories, A-5
- NLS Data Installation Utility, 12-8
- NLS locales, 3-4
- NLS parameters
 - setting, 3-2
 - using in SQL functions, 7-2
- NLS runtime library, 1-2
- NLS_CALEDAR parameter, 3-23
- NLS_CHARSET_DECL_LEN function, 7-9
- NLS_CHARSET_ID function, 7-9
- NLS_CHARSET_NAME function, 7-9
- NLS_COMP parameter, 3-33, 7-11
- NLS_CREDIT parameter, 3-26, 3-31
- NLS_CURRENCY parameter, 3-26
- NLS_DATE_FORMAT parameter, 3-14
- NLS_DATE_LANGUAGE parameter, 3-16
- NLS_DEBIT parameter, 3-31
- NLS_DUAL_CURRENCY parameter, 3-29
- NLS_ISO_CURRENCY parameter, 3-28
- NLS_LANG
 - choosing a locale with, 3-4
 - environment variable, 9-6
 - examples, 3-6
 - specifying, 3-6
- NLS_LANGUAGE parameter, 3-8
- NLS_LIST_SEPARATOR parameter, 3-34
- NLS_MONETARY_CHARACTERS
 - parameter, 3-30
- NLS_NUMERIC_CHARACTERS parameter, 3-25
- NLS_SORT parameter, 3-32, 3-33
- NLS_TERRITORY parameter, 3-10
- NLSSORT function, 7-9
- NLT files, 11-2
- numeric
 - formats, 3-24, 7-13
 - parameters, 3-24

O

- ORA_NLS33 directory, 1-3
- Oracle Real Application Clusters
 - during database character set migration, 10-9
- oracle.sql.CHAR, 9-4
- oracle.sql.CHAR class, 9-8
 - getString() method, 9-8
 - getStringWithReplacement() method, 9-8
 - toString() method, 9-8
- oracle.sql.CLOB, 9-4
- ORANLS option, 12-8
- ORDER BY clause, 7-12
- overriding language and territory specifications, 3-6

P

- parameters
 - BLANK_TRIMMING, 10-4
 - calendar, 3-21
 - collation, 3-31
 - CSM\$COLUMNS, 10-35
 - CSM\$ERRORS, 10-36
 - CSM\$TABLES, 10-35
 - monetary, 3-26
 - NLS_CALENDAR, 3-23
 - NLS_COMP, 3-33
 - NLS_CREDIT, 3-26, 3-31
 - NLS_CURRENCY, 3-26
 - NLS_DATE_FORMAT, 3-14
 - NLS_DATE_LANGUAGE, 3-16
 - NLS_DEBIT, 3-31
 - NLS_DUAL_CURRENCY, 3-29
 - NLS_ISO_CURRENCY, 3-28
 - NLS_LANGUAGE, 3-8
 - NLS_LIST_SEPARATOR, 3-34
 - NLS_MONETARY_CHARACTERS, 3-30
 - NLS_NUMERIC_CHARACTERS, 3-25
 - NLS_SORT, 3-32, 3-33
 - NLS_TERRITORY, 3-10
 - numeric, 3-24
 - setting, 3-2
 - time, 3-14
- partitioned
 - indexes, 7-12

- tables, 7-12

R

- replacement characters, 7-5
- restricted multilingual support, 2-18
- restrictions
 - data truncation, 10-3
 - passwords, 10-3
 - space padding during export, 10-4
 - usernames, 10-3
- RM format element, 3-15
- Roman numerals
 - format mask for, 3-15

S

- scan modes
 - Character Set Scanner Utility, 10-11
 - database character sets
 - full database scan, 10-12
 - single table scan, 10-12
 - user tables scan, 10-12
- sorting
 - specifying non-default, 3-32, 3-33
- space padding
 - during export, 10-4
- SQLJ
 - client, 9-35
- SQLJ translators, 9-3
- storage character sets, A-6
- stored procedures
 - Java, 9-16
- string comparisons
 - and WHERE clause, 7-11
- string manipulation using OCI, 8-8
- subsets
 - and supersets, A-19
- supersets
 - and subsets, A-19
- supported character sets, 2-15
- supported character string functionality and character sets, 2-15
- suppress parameter
 - Character Set Scanner Utility, 10-21

T

- table parameter
 - Character Set Scanner Utility, 10-21
- tables
 - partitioned, 7-12
- territories, 3-10
 - overriding, 3-6
 - supported, A-5
- territory definition
 - setting, 11-11
- territory support, 1-6
- time parameters, 3-14
- TO_CHAR function
 - default date format, 3-14
 - format masks, 7-12
 - group separator, 3-25
 - language for dates, 3-16
 - spelling of days and months, 3-16
- TO_DATE function
 - default date format, 3-14
 - format masks, 7-12
 - language for dates, 3-16
 - spelling of days and months, 3-16
- TO_NUMBER function
 - format masks, 7-12
 - group separator, 3-25
- tochar parameter
 - Character Set Scanner Utility, 10-22
- toString() method, 9-8
- translated messages, A-4
- translators
 - SQLJ, 9-3

U

- Unicode, 5-2
 - character code assignments, B-2
 - data migration, 5-14
- Unicode encoding, 5-2
- user parameter
 - Character Set Scanner Utility, 10-22
- userid parameter
 - Character Set Scanner Utility, 10-23
- UTF-16 encoding, B-3
- UTF8, A-18

- UTF8 character set, 5-5
- UTF-8 encoding, B-3
- UTFE, A-18
- UTFE character set, 5-5

W

- WHERE clause
 - and string comparisons, 7-11