Oracle8*i*

National Language Support Guide

Release 2 (8.1.6)

December 1999

Part No. A76966-01

ORACLE

National Language Support Guide, Release 2 (8.1.6)

Part No. A76966-01

Copyright © 1996, 1999, Oracle Corporation. All rights reserved.

Primary Author: Paul Lane

Contributors: Winson Chu, Jason Durbin, Jessica Fan, Yu Gong, Josef Hasenberger, Claire Ho, Peter Linsley, Tom Portfolio, Den Raphaely, Linus Tanaka, Makoto Tozawa, Gail Yamanaka, 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 Enterprise Manager, Pro*COBOL, Server Manager, SQL*Forms, SQL*Net, and SQL*Plus, Net8, Oracle Call Interface, Oracle7, Oracle7 Server, Oracle8, Oracle8 Server, Oracle8i, Oracle Forms, PL/SQL, Pro*C, Pro*C/C++, and Trusted Oracle are registered trademarks or trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

Contents

2 Setting Up an NLS Environment

3 Choosing a Character Set

4 SQL Programming

5 OCI Programming

6 Java

A Locale Data

B Customizing Locale Data

Send Us Your Comments

Oracle8*i* National Language Support Guide, Release 2 (8.1.6)

Part No. A76966-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

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

- E-mail - infodev@us.oracle.com
- FAX - (650) 506-7228. Attn: Information Development
- Postal service:
  Oracle Corporation
  Server Technologies Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

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

Preface

This manual provides reference information about Oracle's National Language Support (NLS) capabilities. This information includes:

Feature Coverage and Availability

Oracle8i National Language Support Guide describes how to deal with many of the common problems in working in environments with multiple languages or character sets.

> **See Also:** *Getting to Know Oracle8i* for information about the differences between Oracle8*i* and the Oracle8*i* Enterprise Edition and the available features and options. That book also describes all the features that are new in Oracle8*i*. *Oracle8i National Language Support Guide* describes those features which are common to both products.

Audience

This manual is written for database administrators, system administrators, and database application developers who need to deal with NLS-related matters.

Knowledge Assumed of the Reader

It is assumed that readers of this manual are familiar with relational database concepts, basic Oracle server concepts, and the operating system environment under which they are running Oracle.

Installation and Migration Information

This manual is not an installation or migration guide. If your primary interest is installation, refer to your operating-system-specific Oracle documentation. If your primary interest is database and application migration, refer to *Oracle8i Migration*.

Application Design Information

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 *Oracle8i Application Developer's Guide - Fundamentals* and to the documentation for the tool or language product they are using to develop Oracle database applications.

How Oracle8*i* National Language Support Guide Is Organized

This manual is organized as follows:

Chapter 1, "Understanding Oracle NLS"

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

Chapter 2, "Setting Up an NLS Environment"

This chapter contains an explanation of Oracle's NLS capabilities.

Chapter 3, "Choosing a Character Set"

This chapter contains sample scenarios for enabling NLS capabilities.

Chapter 4, "SQL Programming"

This chapter describes NLS considerations for SQL programming.

Chapter 5, "OCI Programming"

This chapter describes NLS considerations for OCI programming.

Chapter 6, "Java"

This chapter describes NLS considerations for Java.

Appendix A, "Locale Data"

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

Appendix B, "Customizing Locale Data"

This chapter shows how to customize NLS data objects.

Appendix C, "Obsolete Locale Data"

This chapter lists some obsolete names for character sets.

Appendix D, "Glossary"

This chapter defines NLS terminology.

Conventions Used in This Manual

The following sections describe the conventions used in this manual.

Text of the Manual

The text of this manual uses the following conventions.

UPPERCASE Characters

Uppercase text is used to call attention to command keywords, database object names, parameters, filenames, and so on.

For example, "After inserting the default value, Oracle checks the FOREIGN KEY integrity constraint defined on the DEPTNO column," or "If you create a private rollback segment, the name must be included in the ROLLBACK_SEGMENTS initialization parameter."

Italicized Characters

Italicized words within text are book titles or emphasized words.

Code Examples

Commands or statements of SQL, Oracle Enterprise Manager line mode, and SQL*Plus appear in a monospaced font.

For example:

```
INSERT INTO emp (empno, ename) VALUES (1000, 'SMITH');
ALTER TABLESPACE users ADD DATAFILE 'users2.ora' SIZE 50K;
```

Example statements may include punctuation, such as commas or quotation marks. All punctuation in example statements is required. All example statements

terminate with a semicolon (;). Depending on the application, a semicolon or other terminator may or may not be required to end a statement.

UPPERCASE in Code Examples

Uppercase words in example statements indicate the keywords within Oracle SQL. When you issue statements, however, keywords are not case sensitive.

`lowercase` in Code Examples

Lowercase words in example statements indicate words supplied only for the context of the example. For example, lowercase words may indicate the name of a table, column, or file.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of our manuals. As we write, revise, and evaluate our documentation, your opinions are the most important feedback we receive.

You can send comments and suggestions about this manual to the Information Development department at the following e-mail address:

infodev@us.oracle.com

If you prefer, you can send letters or faxes containing your comments to:

Server Technologies Documentation Manager
Oracle Corporation
500 Oracle Parkway Redwood Shores, CA 94065
Fax: (650) 506-7228 Attn: National Language Support Guide

1

Understanding Oracle NLS

This chapter provides an overview of Oracle NLS support, including:

- Oracle Server NLS Architecture
- Standard Features
- Customization Features
- SQL Support

Oracle Server NLS Architecture

Oracle's National Language Support (NLS) architecture allows you to store, process, and retrieve data in native languages. It ensures that database utilities and error messages, sort order, date, time, monetary, numeric, and calendar conventions automatically adapt to the native language and locale.

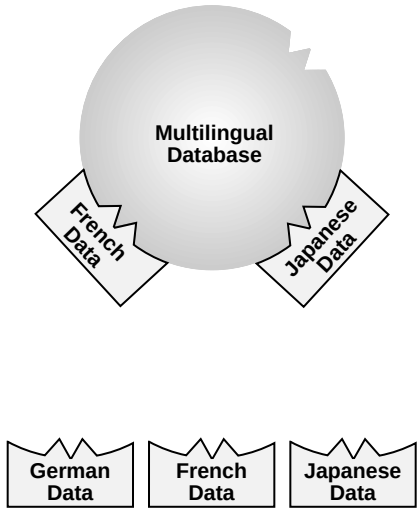
Parameter settings determine the behavior of individual conventions.

Locale-Independent Operation

Oracle's National Language Support architecture is implemented with the use of the Oracle NLS Runtime Library. The NLS Runtime library provides a comprehensive suite of language-independent functions, which allows for 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 run time. For example, French and Japanese locale data is loaded.

Figure 1–1 Loading Locale-specific Data at Runtime



The locale-specific NLS data is stored in a directory specified by the ORA_NLS* environment variable. For each new release, there is a different corresponding ORA_NLS data directory. For Oracle8*i*, the ORA_NLS33 directory is used. For example, on most UNIX platforms, the environment variable ORA_NLS33 should be set to **$ORACLE_HOME/ocommon/nls/admin/data**. On Win32 platforms, the default setting done by the installer should work fine as long as one ORACLE_HOME has just one release of Oracle.

Table 1–1 Location of NLS Data

Release	Environment Variable
7.2	ORA_NLS
7.3	ORA_NLS32
8.0, 8.1	ORA_NLS33

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 data files 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.

Client/Server Architecture

Oracle8*i* is implemented using a client/server architecture. The language-dependent operations are controlled by a number of parameters and environment variables on both the client and the server. On the server, each session started on behalf of a client may run in the same or 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 them are the database and the national character set. They specify the character set used to store text data in the database. Other parameters, like language and territory, are used in the evaluation of CHECK constraints.

In the event that the client and server specify different character sets, Oracle8*i* will handle character set conversion of strings automatically.

As far as NLS architecture is concerned, all applications, even these running on the same physical machine as the Oracle instance, are considered clients. For example, SQL*Plus started by the Unix user which owns Oracle software, from the Oracle Home in which RDBMS software is installed, and connecting to the database through an adapter by specifying the ORACLE_SID, is still considered a client and its behavior is ruled by client-side NLS parameters.

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 Chapter 5, "OCI Programming", for further details.

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 the session NLS environment only. It does not change the local client NLS environment. The session NLS settings are used in the processing of SQL and PL/SQL statements 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 Chapter 2, "Setting Up an NLS Environment", for details.

Standard Features

Oracle's standard features include

- Language Support
- Territory Support
- Date and Time Formats
- Monetary and Numeric Formats
- Calendars
- Linguistic Sorting
- Character Set Support

Language Support

Oracle8*i* allows users to store, process, and retrieve data in native languages. The languages that can be stored in an Oracle8*i* database are all languages written in scripts encoded by Oracle-supported character sets. Through the Unicode (UTF8) character set, Oracle8*i* supports most contemporary languages.

Additional support is available for a subset of the native languages, for which Oracle8*i* knows, for example, how to display dates using translated month names or how to sort text data according to cultural conventions.

When using the term *language support*, this manual 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.

Table 1–2 lists the languages supported, with an asterisk for languages with translated error messages.

Table 1–2 Language Support

American English *	English	Italian *	Russian *
Arabic *	Estonian	Japanese *	Simplified Chinese *
Bengali	Finnish *	Korean *	Slovak *
Brazilian Portuguese *	French *	Latin American Spanish *	Slovenian
Bulgarian	German *	Latvian	Spanish *
Canadian French	German Din	Lithuanian	Swedish *
Catalan *	Greek *	Malay	Tamil
Croatian	Hebrew *	Mexican Spanish	Thai
Czech *	Hindi	Norwegian *	Traditional Chinese *
Danish *	Hungarian *	Polish *	Turkish *
Dutch *	Icelandic	Portuguese *	Ukrainian
Egyptian	Indonesian	Romanian *	Vietnamese

See "Languages" on page A-2 for a complete list of Oracle language names and abbreviations.

Message Support

Utilities and error messages can be made to appear in the native language. See "Translated Messages" on page A-4, for further details.

Territory Support

Oracle8*i* supports different cultural conventions which are specific to a given geographical location. Local time, date, numeric and monetary conventions are handled. The following territories are supported.

Table 1–3 Territory Support

Algeria	Estonia	Latvia	Slovenia
America	Finland	Lebanon	Somalia
Austria	France	Libya	South Africa
Australia	Germany	Lithuania	Spain
Bahrain	Greece	Luxembourg	Sudan
Bangladesh	Hong Kong	Malaysia	Sweden

Table 1–3 Territory Support

Belgium	Hungary	Mauritania	Switzerland
Brazil	Iceland	Mexico	Syria
Bulgaria	India	Morocco	Taiwan
Canada	Indonesia	New Zealand	Thailand
Catalonia	Iraq	Norway	The Netherlands
China	Ireland	Oman	Tunisia
CIS	Israel	Poland	Turkey
Croatia	Italy	Portugal	Ukraine
Cyprus	Japan	Qatar	United Arab Emirates
Czech Republic	Jordan	Romania	United Kingdom
Denmark	Kazakhstan	Saudi Arabia	Uzbekistan
Djibouti	Korea	Singapore	Vietnam
Egypt	Kuwait	Slovakia	Yemen

Date and Time Formats

The world's various conventions for hour, day, month, and year can be handled in local formats.

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.

Calendars

Gregorian, Japanese Imperial, ROC Official, Thai Buddha, Persian, English Hijrah, and Arabic Hijrah are supported. See "Calendar Systems" on page A-21 for a complete list of calendars.

Linguistic Sorting

Oracle8*i* provides linguistic definitions for culturally accurate sorting and case conversion.

Some of the definitions listed in Table 1–4 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 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.

Table 1–4 Linguistic Definitions

Basic Name	Extended Name	Special Cases
ARABIC	--	
ARABIC_MATCH	--	
ARABIC_ABJ_SORT	--	
ARABIC_ABJ_MATCH	--	
ASCII7	--	
BENGALI	--	
BULGARIAN	--	
CANADIAN FRENCH	--	
CATALAN	XCATALAN	æ, AE, ß
CROATIAN	XCROATIAN	D, L, N, d, l, n, ß
CZECH	XCZECH	ch, CH, Ch, ß
DANISH	XDANISH	A, ß, Å , å
DUTCH	XDUTCH	ij, IJ
EEC_EURO	--	
EEC_EUROPA3	--	
ESTONIAN	--	
FINNISH	--	
FRENCH	XFRENCH	
GERMAN	XGERMAN	ß
GERMAN_DIN	XGERMAN_DIN	ß, ä, ö, ü, Ä, Ö, Ü
GREEK	--	
HEBREW	--	
HUNGARIAN	XHUNGARIAN	cs, gy, ny, sz, ty, zs, ß, CS, Cs, GY, Gy, NY, Ny, SZ, Sz, TY, Ty, ZS, Zs

Table 1–4 Linguistic Definitions

Basic Name	Extended Name	Special Cases
ICELANDIC	--	
INDONESIAN	--	
ITALIAN	--	
JAPANESE	--	
LATIN	--	
LATVIAN	--	
LITHUANIAN	--	
MALAY	--	
NORWEGIAN	--	
POLISH	--	
PUNCTUATION	XPUNCTUATION	
ROMANIAN	--	
RUSSIAN	--	
SLOVAK	XSLOVAK	dz, DZ, Dz, ß (*caron*)
SLOVENIAN	XSLOVENIAN	ß
SPANISH	XSPANISH	ch, ll, CH, Ch, LL, Ll
SWEDISH	--	
SWISS	XSWISS	ß
THAI_DICTIONARY	--	
THAI_TELEPHONE	--	
TURKISH	XTURKISH	æ, AE, ß
UKRAINIAN	--	
UNICODE_BINARY		
VIETNAMESE	--	
WEST_EUROPEAN	XWEST_EUROPEAN	ß

Character Set Support

Oracle supports a large number of single-byte, multi-byte, and fixed-width encoding schemes which are based on national, international, and vendor-specific

standards. See "Character Sets" on page A-6 for a complete list of supported character sets.

Customization Features

Oracle allows you to customize character sets and calendars.

Character Set Customization

User-defined characters are sometimes needed to support special symbols, vendor-specific characters, or characters that represent proper names, historical terms, and so on. Developers can extend an existing character set definition by using the Unicode Private Use Area. See "Customized Character Sets" on page B-2 for further information.

Calendar Customization

You can define ruler eras for imperial calendars, and deviation days for lunar calendars. See "Customized Calendars" on page B-11 for further information.

SQL Support

NLS parameters can be used to modify the behavior of SQL functions. For instance, SQL functions that deal with time, date, monetary, and numeric formats, as well as sorting and character classification, can change behavior based on different NLS parameters that are implicitly set in the users' environment or explicitly set as a parameter to a function call. See Chapter 4, "SQL Programming", for further information about function calls and see Chapter 2, "Setting Up an NLS Environment", for information about environment parameters.

2

Setting Up an NLS Environment

This chapter tells how to set up an NLS environment, and includes the following topics:

Setting NLS Parameters

NLS parameters determine the locale-specific behavior on both the client and the server. There are four ways to specify NLS parameters:

1. 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"
   ```

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

   ```
   % setenv NLS_SORT FRENCH
   ```

3. As **ALTER SESSION parameters**. NLS parameters set in an ALTER SESSION statement can be used to override the defaults set for the session in the initialization file, or set by the client with environment variables.

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

   For a complete description of ALTER SESSION, see *Oracle8i SQL Reference*.

4. As a **SQL function parameter**. NLS parameters can be used explicitly to hardcode NLS behavior within a SQL function. Doing so will override the defaults set for the session in the initialization file, the client with environment variables, or set for the session by ALTER SESSION. 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. For a complete description of CREATE DATABASE, see *Oracle8i SQL Reference*.

Table 2–1 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. And explicitly setting an NLS parameter within a SQL function can override all other settings — default, initialization parameter, environment variable, and ALTER SESSION parameters.

Table 2–1 Parameters and Their Priorities

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

Table 2–2 lists the NLS parameters available with the Oracle server.

Table 2–2 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 Operator comparison	Binary	I, E, A
NLS_CREDIT	Credit accounting symbol	NLS_TERRITORY	-, E, -
NLS_CURRENCY	Local currency symbol	NLS_TERRITORY	I, E, A
NLS_DATE_FORMAT	Date format	NLS_TERRITORY	I, E, A
NLS_DATE_LANGUAGE	Language for day and month names	NLS_LANGUAGE	I, E, A
NLS_DEBIT	Debit accounting symbol	NLS_TERRITORY	-, E, -
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_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, -

Table 2–2 Parameters and their Scope

NLS_NCHAR	National character set	NLS_LANG	-, 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_DUAL_CURRENCY	Dual currency symbol	NLS_TERRITORY	I, E, A

Choosing a Locale with NLS_LANG

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, i.e., the character set of data entered or displayed by a client program.

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

```
NLS_LANG = language_territory.charset
```

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

language	Specifies conventions such as the language used for Oracle messages, collation, 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, so either (or both) territory or charset can be omitted. If language is not specified, the value defaults to American. For a complete list of languages, see Appendix A, "Locale Data".
territory	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. For a complete list of territories, see Appendix A, "Locale Data".
charset	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 could be set, but would not work properly. For example, the following tries to support Japanese by using a Western European character set:

```
NLS_LANG = JAPANESE_JAPAN.WE8DEC
```

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

Specifying NLS_LANG

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

```
% 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 NLS_LANG.

```
% 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, the same query returns:

```
% 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

NLS_LANG 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 NLS_LANG set the default values for the other NLS parameters, such as date format, numeric characters, and collation. Each of these detailed parameters can be set in the client environment to fine-tune the language and territory defaults.

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

If NLS_LANG is not set, the server session environment remains initialized with values of NLS_LANGUAGE, NLS_TERRRITORY, 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 does it; Oracle Developer is an example of an application that does not do this.

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 settings with some views and 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 file (INIT.ORA) 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 *Oracle8i Reference* for further details.

OCI Functions

To allow user applications to query client NLS settings Oracle8i OCI contains the `OCINlsGetInfo` function. See Chapter 5, "OCI Programming", for the description of this function.

Language and Territory Parameters

NLS_LANGUAGE and NLS_TERRITORY parameters are general NLS parameters describing NLS behavior of locale-dependent operations.

NLS_LANGUAGE

Parameter type:	String
Parameter scope:	Initialization Parameter and ALTER SESSION
Default value:	Derived from NLS_LANG
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
- 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 / negative response strings

The value specified for NLS_LANGUAGE in the initialization 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 ORA_RDBMS 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 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.

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

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_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 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 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 initialization file value is overridden already at the 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, the following statement should be issued:

```
SQL> 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

SQL> ALTER SESSION SET NLS_TERRITORY = Germany;
Session altered.

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

SQL> ALTER SESSION SET NLS_LANGUAGE = German;
Sitzung wurde geändert.

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

SQL> 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

SQL> ALTER SESSION SET NLS_LANGUAGE = German
2     > NLS_DATE_FORMAT = 'DD.MON.YY'
3     > 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.

Time Parameters

Many different time formats are used throughout the world. Some typical ones are shown in Table 2–3.

Table 2–3 Time Parameters

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:ss am	1.50.23 PM

Date Parameters

Oracle allows you to control how dates appear through the use of date parameters.

Date Formats

Many different date formats are used throughout the world. Some typical ones are shown in Table 2–4.

Table 2–4 Date Formats

Country	Description	Example
Estonia	dd.mm.yyyy	28.02.1998

Table 2–4 Date Formats

Country	Description	Example
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.

Year 2000 Issues

Currently, the default date format for most territories specifies the year format as "RR" to indicate the last 2 digits. If your applications are Year 2000 compliant, you can safely specify the NLS_DATE_FORMAT using "YYYY" or "RRRR". If your applications are not yet Year 2000 compliant, you may wish to specify the NLS_DATE_FORMAT as "RR". The "RR" format will have the following effect: Given a year with 2 digits, RR will return a year in the next century if the year is less than 50 and the last 2 digits of the current year are greater than or equal to 50; return a year in the preceding century if the year is greater than or equal to 50 and the last 2 digits of the current year are less than 50.

See the Date Format Models section in the *Oracle8i SQL Reference* for full details on Date Format Elements.

Date Formats and Partition Bound Expressions

Partition bound expressions for a date column must specify a date using a format which requires that the month, day, and 4-digit year are 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 current century (that is, if your default date format is MM-DD-YY for example), 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.

For more information on using TO_DATE(), see *Oracle8i SQL Reference*.

NLS_DATE_LANGUAGE

Parameter type:	String
Parameter scope:	Initialization Parameter, Environment Variable, and ALTER SESSION
Default value:	Derived from NLS_LANGUAGE
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

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

returns

```
Mercredi:12 Février 1997
```

Month and day name abbreviations are also in the language specified, for example:

```
SQL> 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, the above date would be inserted using:

```
SQL> 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:

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

returns:

```
Mercredi: twelfth Février
```

You can alter the default value of NLS_DATE_LANGUAGE 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_LANGUAGE statement.

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 2–5 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, a format element IW is provided that returns the ISO week number.

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

Table 2–6 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 2–7 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
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, as well as 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 2–8.

Table 2–8 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.

Note: 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 file and then restart the instance.
- Use the ALTER SESSION SET NLS_NUMERIC_CHARACTERS command 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 2–9.

Table 2–9 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

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

would return

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

You can alter the default value of NLS_CURRENCY by changing its value in the initialization file and then restarting the instance, and 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 that 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 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

```
SQL> 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 file and then restarting the instance, and 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 2–10.

Table 2–10 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.

NLS_DUAL_CURRENCY was introduced to help support the Euro. The following Table 2–11 lists the character sets that support the Euro symbol:

Table 2–11 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

Table 2–11 Character Sets that Support the Euro Symbol

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
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
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 520-2533 - ASCII 8-bit	0x80
AL24UTFFSS	Unicode 1.1 UTF-8 Universal character set	U+20AC
UTF8	Unicode 2.1 UTF-8 Universal character set	U+20AC
UTFE	UTF-EBCDIC encoding of Unicode 2.1	U+20AC

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.

Collation Parameters

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

Sorting Order

Different languages have different sort orders. What's more, different cultures or countries using the same alphabets may sort words differently. For example, the German language sharp s (ß) is sorted differently in Germany and Austria. The linguistic sort sequence *German* sorts this sequence as the two characters (SS), while the linguistic sort sequence *Austrian* sorts it as (SZ). Another example is the treatment of *ö*, *o*, and *œ*. They are sorted differently throughout the various Germanic languages.

Oracle provides many different types of sort, 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*. But a true linguistic sort will cause some performance degradation.

Sorting East Asian languages is difficult and complex. At present, Oracle typically relies on the binary order of the particular encoded character set for sorting East Asian Languages.

Sorting Character Data

Conventionally, when character data is sorted, the sort sequence is based on the numeric values of the characters defined by the character encoding scheme. Such a sort is called a *binary* sort. Such a sort produces 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.

Binary Sorts

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 that sequence, when the Ä has a higher numeric value than B in the character encoding scheme.

Linguistic Sorts

To produce a sort sequence that matches the alphabetic sequence of characters for a particular language, 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 other binary values that reflect the character's proper linguistic order so that a sort returns the desired result.

The Oracle server provides both sort mechanisms. Linguistic sort sequences are defined as part of language-dependent data. Each linguistic sort sequence has a unique name. NLS parameters define the sort mechanism for ORDER BY queries. A default value can be specified, and this value can be overridden for each session with the NLS_SORT parameter. A complete list of linguistic definitions is provided in "Linguistic Definitions" on page A-19.

Warning: Linguistic sorting is not supported on Asian multi-byte character sets. If the database character set is multi-byte, you will get binary sorting, which makes the sort sequence dependent on the character set specification. There are two exceptions to this rule: Japanese Hiragana/Katakana and the UTF8 character set. This means that the Japanese *Yomi* sort is only possible by creating an extra column using the Hiragana or Katakana reading for the kanji and sorting on that column.

Linguistic Indexes

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 just improves the performance. A simple example is:

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

So

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

returns the result much faster than without an index.

For more information, see the description of function-based indexes in *Oracle8i Concepts*.

Multiple Linguistic Indexes

If you store character data of multiple languages into one database, you may want to create multiple linguistic indexes for one column. This approach improves the performance of the linguistic sort for a specific column for multiple languages and is a powerful feature for multilingual databases.

An example of creating multiple linguistic indexes is:

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

When session variable NLS_SORT is set to FRENCH, french_index can be used and when it is set to GERMAN, german_index can be used.

Requirements for Linguistic Indexes

If you want to use a single linguistic index or multiple linguistic indexes, there are some requirements to be met in order for the linguistic index to be used. The first requirement is that QUERY_REWRITE_ENABLED needs to be 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 ANSI. There are various ways to set NLS_COMP.

Here is an 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 FRENCH. If you want a GERMAN linguistic sort order, NLS_SORT needs to be GERMAN. There are various ways to set NLS_SORT. The below is an example. Although the example below uses the ALTER SESSION statement, it's 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 picked up based on NLS_SORT being set in the client environment.

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

The fourth requirement is that you need to use the cost-based optimizer. Function-based indexes do not get picked up by the rule-based optimizer. Function-based indexes are used only by the cost-based optimizer.

The last thing is that you need to specify "WHERE NLSSORT(column_name) IS NOT NULL" when you want to use "ORDER BY column_name" and the "column_name" is the column with the linguistic index. This is necessary only when you use "ORDER BY". See the example below.

The below is an example of using a FRENCH linguistic index. For NLS_SORT, you may want to set it in the client environment variable instead of the ALTER SESSION statement (as described above).

```
ALTER SESSION SET query_rewrite_enabled=true;
ALTER SESSION SET NLS_COMP = ANSI;
ALTER SESSION SET NLS_SORT='FRENCH';
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';
```

For more information, see the description of function-based indexes in *Oracle8i Concepts*.

Case-Insensitive Search

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

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

For more information, see the description of function-based indexes in *Oracle8i Application Developer's Guide - Fundamentals*.

Linguistic Special Cases

Linguistic special cases are character sequences that need to be treated as a single character when sorting. Such special cases are handled automatically when using a linguistic sort. For example, one of the linguistic sort sequences for Spanish specifies that the double characters *ch* and *ll* are sorted as single characters appearing between *c* and *d* and between *l* and *m* respectively.

Another example is the German language sharp s (ß). The linguistic sort sequence *German* can sort this sequence as the two characters *SS*, while the linguistic sort sequence *Austrian* sorts it as *SZ*.

Special cases like these are also handled when converting uppercase characters to lowercase, and vice versa. For example, in German the uppercase of the sharp s (ß) is the two characters *SS*. 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 do not handle these special cases.)

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). But 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 file and then restarting the instance, and you can alter its value during a session using an ALTER SESSION SET NLS_SORT command.

A complete list of linguistic definitions is provided in Table A–8, "Linguistic Definitions".

NLS_COMP

Parameter type:	String

Parameter scope:	Initialization Parameter, Environment Variable and ALTER SESSION
Default value:	Binary
Range of values:	BINARY or ANSI

You can use this parameter to avoid the cumbersome process of using NLS_SORT in SQL statements. Normally, comparison in the WHERE clause 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:

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

To specify that comparison in the WHERE clause is always binary, issue

```
SQL> 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:

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

NLS_LIST_SEPARATOR

Parameter type:	String
Parameter scope:	Environment Variable
Default value:	Derived from NLS_TERRITORY
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 Parameters

You can specify the character set used for the client.

NLS_NCHAR

Parameter type:	String
Parameter scope:	Environment Variable
Default value:	Derived from NLS_LANG
Range of values:	Any valid character set name

NLS_NCHAR specifies the character set used by the client application for national character set data (NCHAR, NVARCHAR2, NCLOB). If it is not specified, the client application uses the same character set that it uses for the database character set data.

3

Choosing a Character Set

This chapter explains NLS topics that you need to know when choosing a character set. These topics are:

What is an Encoded Character Set?

An encoded character set is specified when creating a database, and your choice of character set determines what languages can be represented in the database. This choice also 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.

When processing characters, computer systems handle character data as numeric codes rather than as their graphical representation. For instance, when the database stores the letter "A", it actually stores a numeric code that is interpreted by software as that letter.

A group of characters (for example, alphabetic characters, ideographs, symbols, punctuation marks, control characters) can be encoded as a coded character set. A coded character set assigns unique numeric codes to each character in the character repertoire. Table 3–1 shows examples of characters that are assigned a numeric code value.

Table 3–1 Encoded Characters in the ASCII Character Set

Character	Description	Code Value
!	Exclamation Mark	0x21
#	Number Sign	0x23
$	Dollar Sign	0x24
1	The Number 1	0x31
2	The Number 2	0x32
3	The Number 3	0x33
A	An Uppercase A	0x41
B	An Uppercase B	0x42
C	An Uppercase C	0x43
a	A Lowercase a	0x61
b	A Lowercase b	0x62
c	A Lowercase c	0x63

There are many different coded character sets used throughout the computer industry and supported by Oracle. Oracle supports most national, international, and vendor-specific encoded character set standards. The complete list of character

sets supported by Oracle is included in Appendix A, "Locale Data". Character sets differ in:

- the number of characters available
- the particular characters (character repertoire) available
- the writing script(s) and the languages therefore represented
- the code values assigned to each character in the repertoire
- the encoding scheme used to represent a character entity

These differences are discussed throughout this chapter.

Which Characters to Encode?

The first choice to make when choosing a character set is based on 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 broad categories, phonetic and ideographic.

Phonetic Writing Systems

Phonetic writing systems consist of symbols which 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 since the character repertoire is usually smaller than 256 characters.

Ideographic Writing Systems

Ideographic writing systems, in contrast, 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 must typically be encoded in more than one byte because the character repertoire can be as large as 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, apostrophes), numbers (for example, Arabic digits 0-9), special symbols (for example, currency symbols, math operators) and control characters for computers (for example, carriage returns, tabs, 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. 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". Irrespective 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 representable in all Oracle database character sets and transportable to any platform:

- Upper and lower case 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 or
- the national character set feature (NCHAR or NVARCHAR characters)

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 store any values other than well-formed strings in RAW columns rather than CHAR or VARCHAR columns.

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 3–2 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 3–2 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 based on many national or regional standards to support different European languages.

Table 3–3 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, Luxemburgish, 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, Luxemburgish, 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 8859-15	Western European (Albanian, Basque, Breton, Catalan, Danish, Dutch, English, Estonian, Faroese, Finnish, French, Frisian, Galician, German, Greenlandic, Icelandic, Irish Gaelic, Italian, Latin, Luxemburgish, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, Swedish)

Character sets evolved and provided restricted multilingual support, 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 39,000 characters and continues to grow.

How are These Characters Encoded?

Different types of encoding schemes have been created by the computer industry. These schemes have different performance characteristics, and can influence your database schema and application development requirements for handling character data, so you need to be aware of the characteristics of the encoding scheme used by the character set you choose. The character set you choose will typically use one of the following types of 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 being ISO 8859-1, which supports many Western European languages.

Figure 3–1 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	ð
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	Õ	å	õ
6	ACK	SYN	&	6	F	V	f	v	¦	¶	Æ	Ö	æ	ö
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	ú
B	VT	ESC	+	;	K	[	k	{	«	»	Ë	Û	ë	û
C	NP	FS	,	<	L	\	l	|	¬	¼	Ì	Ü	ì	ü
D	CR	GS	-	=	M	]	m	}		½	Í	Ý	í	ý
E	SO	RS	.	>	N	^	n	~	®	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 since 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 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 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 part of 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 will be 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_char><standard_name>[S] [C]
[FIXED]
```

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

For instance:

- 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 sometimes used to help differentiate character sets that can only be used on the server (S) or client (C).

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

The optional "FIXED" at the end of the character set name is used to denote a fixed-width multibyte encoding.

Tips on Choosing an Oracle Database Character Set

Oracle uses the database character set for:

- data stored in CHAR, VARCHAR2, CLOB, and LONG columns
- identifiers such as table names, column names, and PL/SQL variables
- entering and storing SQL and PL/SQL program source

Four considerations you should make when choosing an Oracle character set for the database are:

1. What languages does the database need to support?
2. Interoperability with system resources and applications
3. Performance implications
4. 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 obviate 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, the ISO character sets for instance, are also listed explicitly by language. You may want to see the actual characters that are encoded in some cases. The actual code pages are not listed in this manual, however, since most 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 instance, the operating system supplies fonts that correspond to the character set you have chosen. Input methods that support the language(s) desired 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, Oracle can handle character set conversion from the database character set to the operating system 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 any possible data loss.

Also note that 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 "?" 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, using the same character sets for the client and the server can avoid character set conversion, and thus optimize performance.

Database Schema

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 out 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.

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 multi-byte character sets, and they also are the most efficient in terms of space requirements.

Restrictions

You cannot currently choose an Oracle database character set that is a fixed-width multibyte character set. In particular, the following character sets cannot be used as the database character set:

Table 3–4 Restricted Character Sets

JA16EUCFIXED
ZHS16GBKFIXED

Table 3–4 Restricted Character Sets

JA16DBCSFIXED
KO16DBCSFIXED
ZHS16DBCSFIXED
JA16SJISFIXED
ZHT32TRISFIXED
KO16KSC5601FIXED
ZHS16CGB231280FIXED
ZHT32EUCFIXED
ZHT16BIG5FIXED
ZHT16DBCSFIXED

Tips on Choosing an Oracle NCHAR Character Set

In some cases, you may wish to have the ability 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, or to facilitate ease-of-programming. In particular, the following data types can be used with an alternate character set:

- NCHAR
- NVARCHAR2
- NCLOB

Specifying an NCHAR character set allows you to specify an alternate character set from the database character set for use in NCHAR, NVARCHAR2, and NCLOB columns. This can be particularly useful for customers using a variable-width multibyte database character set because NCHAR has the capability to support fixed-width multibyte encoding schemes, whereas the database character set cannot. The benefits in using a fixed-width multibyte encoding over a variable-width one are:

- optimized string processing performance on NCHAR, NVARCHAR2, and NCLOB columns
- ease-of-programming with a fixed-width multibyte character set as opposed to a variable-width multibyte character set

When choosing an NCHAR character set, you must ensure that the NCHAR character repertoire is equivalent to or a subset of the database character set repertoire.

Note: all SQL commands will use the database character set, not the NCHAR character set. Therefore, literals can only be specified in the database character set.

Database Schema

When using the NCHAR, NVARCHAR2, and NCLOB data types, the width specification can be in terms of bytes or characters depending on the encoding scheme used. If the NCHAR character set uses a variable-width multibyte encoding scheme, the width specification refers to bytes. If the NCHAR character set uses a fixed-width multibyte encoding scheme, the width specification will be in characters. For example, NCHAR(20), using the variable-width multibyte character set JA16EUC, will allocate 20 bytes while NCHAR(20) using the fixed-width multibyte character set JA16EUCFIXED will allocate 40 bytes.

Performance Implications

Some string operations are faster when you choose a fixed-width character set for the national character set. For instance, string-intensive operations such as the SQL LIKE operator used on an NCHAR fixed-width column outperform LIKE operations on a multi-byte CHAR column. A possible usage scenario is as follows:

With a Database Character Set of

JA16EUC

Use an NCHAR Character Set of

JA16EUCFIXED

Recommendations

Because SQL text such as the literals in SQL statements can only be represented by the database character set, and not the NCHAR character set, you should choose an NCHAR character set that either has an equivalent or subset character repertoire of the database character set.

Considerations for Different Encoding Schemes

Keep the following points in mind when dealing with encoding schemes.

Be Careful when Mixing Fixed-Width and Varying-Width Character Sets

Because fixed-width multi-byte character sets are measured in characters, and varying-width character sets are measured in bytes, be careful if you use a fixed-width multi-byte character set as your national character set on one platform and a varying-width character set on another platform.

As an example, if you use %TYPE or a named type to declare an item on one platform using the declaration information of an item from the other platform, you might receive a constraint limit too small to support the data. So, for example, "NCHAR (10)" on the platform using the fixed-width multi-byte set allocates enough space for 10 characters, but if %TYPE or the use of a named type creates a correspondingly typed item on the other platform, it allocates only 10 bytes. Usually, this is not enough for 10 characters. To be safe:

- Do not mix fixed-width multi-byte and varying-width character sets as the national character set on different platforms.
- If you do mix fixed-width multi-byte and varying-width character sets as the national character set on different platforms, use varying-length type declarations with relatively large constraint values.

Storing Data in Multi-Byte Character Sets

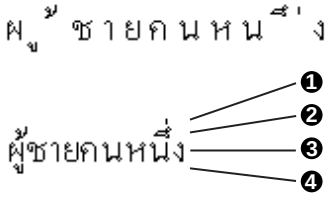
Width specifications of the character datatypes CHAR and VARCHAR2 refer to bytes, not characters. Hence, the specification CHAR(20) in a table definition allows 20 bytes for storing character data.

If the database character set is single byte, and that character set includes only composite characters, the number of characters and the number of bytes are the same. If the database character set is multi-byte, in general, there is no such correspondence. A character can consist of one or more bytes, depending on the specific multi-byte encoding scheme and whether *shift-in/shift-out* control codes are present. Hence, column widths must be chosen with care to allow for the maximum possible number of bytes for a given number of characters.

A typical situation is when character elements are combined to form a single character. For example, *o* and an umlaut can be combined to form ö. In the Thai language, up to three separate character elements can be combined to form one character, and one Thai character would require up to 3 bytes when TH8TISASCII or another single-byte Thai character set is used. One Thai character would require up to 9 bytes when the UTF8 character set is used.

One Thai character consists of up to three separate character elements as shown in Figure 3–2, where two of the characters are comprised of three character elements.

Figure 3–2 Combining Characters



In the lower row of Figure 3–2, nine Thai characters are shown in the correct display format. Inside the database, these nine Thai characters are stored just like the upper row of Figure 3–2. They look like thirteen characters, but they are actually nine characters. Note that the upper row is just showing how Thai characters are stored in the database (and it is the same as how Thai characters are represented in computer memory), but the way shown in the upper row is an incorrect way of displaying Thai characters.

When using the NCHAR and NVARCHAR2 data types, the width specification refers to characters when the national character set is fixed-width multi-byte. Otherwise, the width specification refers to bytes.

A separate performance issue is space efficiency (and thus speed) when using smaller-width character sets. These issues potentially trade-off against each other when the choice is between a varying-width and a fixed-width character set.

Naming Database Objects

You can use Oracle to name database objects.

Restrictions on Character Sets Used to Express Names and Text

Table 3–5 lists the restrictions on the character sets that can be used to express names and other text in Oracle.

Table 3–5 Restrictions on Character Sets Used to Express Names and Text

Name	Single-Byte Fixed	Varying Width	Multi-Byte Fixed Width Character Sets	Comments
Column Names	Yes	Yes	No	
Schema Objects	Yes	Yes	No	
comments	Yes	Yes	No	
database link names	Yes	No	No	
database names	Yes	No	No	
filenames (datafile, logfile, controlfile, initialization parameter file)	Yes	No	No	
instance names	Yes	No	No	
directory names	Yes	No	No	
keywords	Yes	No	No	Can be expressed in English ASCII or EBCDIC characters only
recovery manager filenames	Yes	No	No	
rollback segment names	Yes	No	No	The ROLLBACK_SEGMENTS parameter does not support NLS
stored script names	Yes	Yes	No	
tablespace names	Yes	Yes	No	

For a list of supported string formats and character sets, including LOB data (LOB, BLOB, CLOB, and NCLOB), see Table 3–7.

The character encoding scheme used by the database is defined at database creation as part of the CREATE DATABASE statement. All data columns of type 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. Data columns of type 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 set(s) 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 (or equivalent) 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. Similarly, the character set used for national character set data is defined by the NLS_NCHAR parameter.

Summary of Data Types and Supported Encoding Schemes

Table 3–6 lists the supported encoding schemes associated with different data types.

Table 3–6 Supported Encoding Schemes for Data Types

Data Type	Single-Byte	Multi-byte Varying Width	Multi-byte Fixed Width
CHAR	Yes	Yes	No
NCHAR	Yes	Yes	Yes
BLOB	Yes	Yes	Yes
CLOB	Yes	Yes	No
NCLOB	Yes	Yes	Yes

Table 3–7 lists the supported data types associated with Abstract Data Types (ADT).

Table 3–7 Supported Data Types for Abstract Data Types

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 instance, 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.

The target character set is a strict superset if and only if each and every codepoint in the source character set is available in the target character set, with the same corresponding codepoint value. For instance, the following migration scenarios can take advantage of the ALTER DATABASE CHARACTER SET statement because US7ASCII is a strict subset of WE8ISO8859P1, ZHS16GBK, and UTF8:

Table 3–8 Sample Migration Scenarios

Current Character Set	New Character Set	New Character Set is Strict Superset?
US7ASCII	WE8ISO8859P1	Yes
US7ASCII	ZHS16GBK	Yes
US7ASCII	UTF8	Yes

Attempting to change the database character set to a character set that is not a strict superset can result in data loss and data corruption. To ensure data integrity, whenever migrating to a new character set that is not a strict superset, you must use export/import. It is essential to do a full backup of the database before using the ALTER DATABASE [NATIONAL] CHARACTER SET statement, since the command cannot be rolled back. 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 WE8ISO8859P1;
```

To change the database character set, perform the following steps. Not all of them are absolutely necessary, but they are highly recommended:

```
SQL> SHUTDOWN IMMEDIATE;   -- or NORMAL
    <do a full backup>

SQL> STARTUP MOUNT;
SQL> ALTER SYSTEM ENABLE RESTRICTED SESSION;
SQL> ALTER SYSTEM SET JOB_QUEUE_PROCESSES=0;
SQL> ALTER DATABASE OPEN;
SQL> ALTER DATABASE CHARACTER SET <new_character_set_name>;
SQL> SHUTDOWN IMMEDIATE;   -- or NORMAL
SQL> 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.

Customizing Character Sets

In some cases, you may wish to tailor a character set to meet specific user needs. In Oracle8*i*, users can extend an existing encoded character set definition to suit their needs. User-defined Characters (UDC) are often used to encode special characters representing:

- Proper names
- Historical Han characters which are not defined in an existing character set standard
- Vendor-specific characters
- New symbols or characters you define

This section describes how Oracle supports UDC. It describes:

- Character Sets with User-Defined Characters
- Oracle's Character Set Conversion Architecture
- Unicode 2.1 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 UDC as follows:

Table 3–9 Shift JIS Codepoint Example

Japanese Shift JIS UDC Range	Number of Codepoints
0xf040-0xf07e, 0xf080-0xf0fc	188
0xf140-0xf17e, 0xf180-0xf1fc	188
0xf240-0xf27e, 0xf280-0xf2fc	188
0xf340-0xf37e, 0xf380-0xf3fc	188
0xf440-0xf47e, 0xf480-0xf4fc	188
0xf540-0xf57e, 0xf580-0xf5fc	188
0xf640-0xf67e, 0xf680-0xf6fc	188
0xf740-0xf77e, 0xf780-0xf7fc	188
0xf840-0xf87e, 0xf880-0xf8fc	188
0xf940-0xf97e, 0xf980-0xf9fc	188

The Oracle character sets listed in Table 3–10 contain pre-defined ranges that allow you to support User Defined Characters:

Table 3–10 Oracle Character Sets with UDC

Character Set Name	Number of UDC Codepoints Available
JA16DBCS	4370
JA16DBCSFIXED	4370
JA16EBCDIC930	4370
JA16SJIS	1880
JA16SJISFIXED	1880
JA16SJISYEN	1880
KO16DBCS	1880
KO16DBCSFIXED	1880

Table 3–10 Oracle Character Sets with UDC

KO16MSWIN949	1880
ZHS16DBCS	1880
ZHS16DBCSFIXED	1880
ZHS16GBK	2149
ZHS16GBKFIXED	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 3–3 Kanji Example

is encoded as follows in different Japanese character sets:

Table 3–11 Kanji Example with Character Conversion

Character Set	Unicode	JA16SJIS	JA16EUC	JA16DBCS
Character Value of 	0x4E9C	0x889F	0xB0A1	0x4867

In Oracle, all character sets are defined in terms of a Unicode 2.1 code point. That is each character is defined as a Unicode 2.1 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 3–3, "Kanji Example" (value 0x889F) entered from the JA16SJIS client is internally converted to Unicode (value 0x4E9C), and then converted to JA16EUC(value 0xB0A1).

Unicode 2.1 Private Use Area

Unicode 2.1 reserves the range 0xE000-0xF8FF for the Private Use Area (PUA). The PUA is intended for private use character definition by end users or vendors.

UDC can be converted between two Oracle character sets by using Unicode 2.1 PUA as the intermediate form, the same as standard characters.

UDC Cross References

UDC 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 UDC on both a Japanese Shift-JIS operating system and a Japanese IBM Host operating system, you may want to pick up 0xF040 on Shift-JIS operating system and 0x6941 on IBM Host operating system for the new UDC so that Oracle can convert correctly between JA16SJIS and JA16DBCS. You can find out that both Shift-JIS UDC value 0xF040 and IBM Host UDC value 0x6941 are mapped to the same Unicode PUA value 0xE000 in the UDC cross reference.

For further details on how to customize a character set definition file, see Appendix B, "Customizing Locale Data".

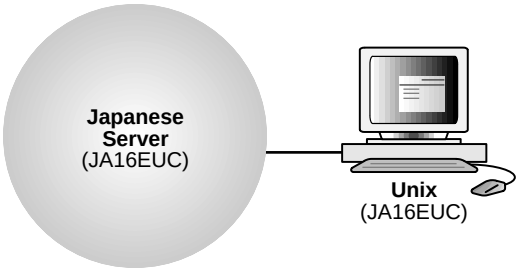
Monolingual Database Example

Same Character Set on the Client and the Server

This section describes the simplest example of an NLS database setup.

Both the client and server in Figure 3–4, "Monolingual Scenario", are running with the same language environment, and are both using the same character encoding. The monolingual scenario has the advantage of fast response because the overhead associated with character set conversion is avoided.

Figure 3–4 Monolingual Scenario

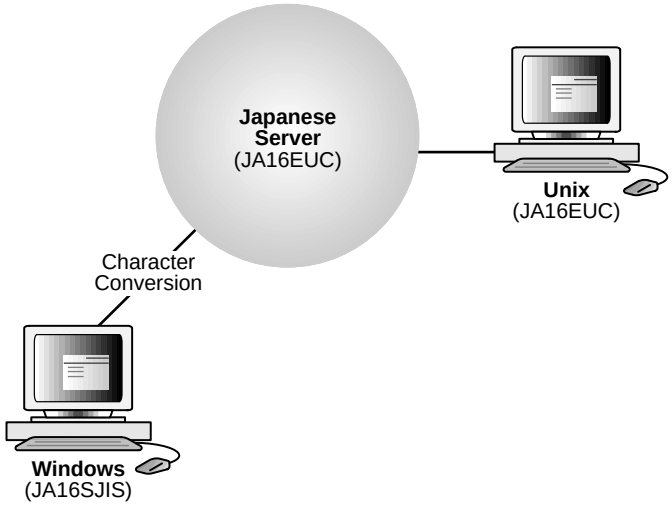


Character Set Conversion

Character set conversion is often necessary in a client/server computing environment where a client application may reside on a different computer platform from that of the server, and both platforms may 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 Net8.

A conversion is possible between any two character sets, as shown in Figure 3–5:

Figure 3–5 Character Set Conversion Example



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 the character *ä* is replaced with *a*.

Replacement characters may be defined for specific characters as part of a character set definition. Where 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 3–4, "Monolingual Scenario", the server's character set was not chosen wisely. If German data is expected to be stored on the server, a character set which supports German letters is needed, for example, WE8ISO8859P1 for both the server and the client.

In some varying-width multi-byte cases, character set conversion may introduce noticeable overhead. Users need to carefully evaluate their 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 any possible data loss.

Multilingual Database Example

Note that some character sets support multiple languages. For example, WE8ISO8859P1 supports the following Western European languages:

Table 3–12 WE8ISO8859P1 Example

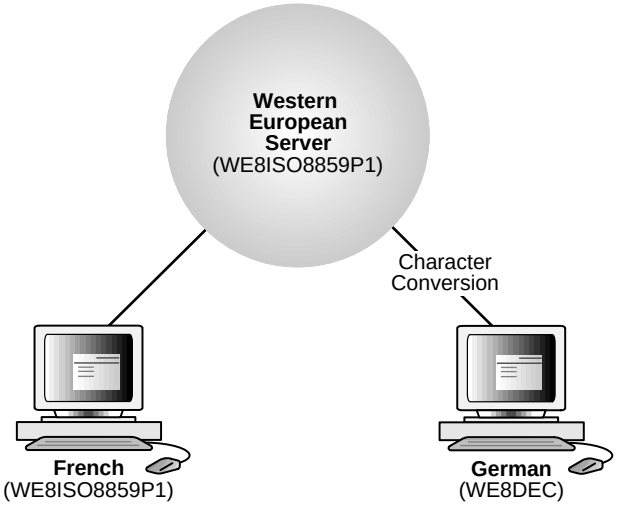
Catalan	Finnish	Italian	Swedish
Danish	French	Norwegian	
Dutch	German	Portuguese	
English	Icelandic	Spanish	

The reason WE8ISO8859P1 supports the languages above is because they are all based on a similar writing script. This situation is often called *restricted* multilingual support. Restricted because this character set supports a group of related writing systems or scripts. In Table 3–12, WE8ISO8859-1 supports Latin-based scripts.

Restricted Multilingual Support

In Figure 3–6, both clients have access to the server's data.

Figure 3–6 Restricted Multilingual Support Example



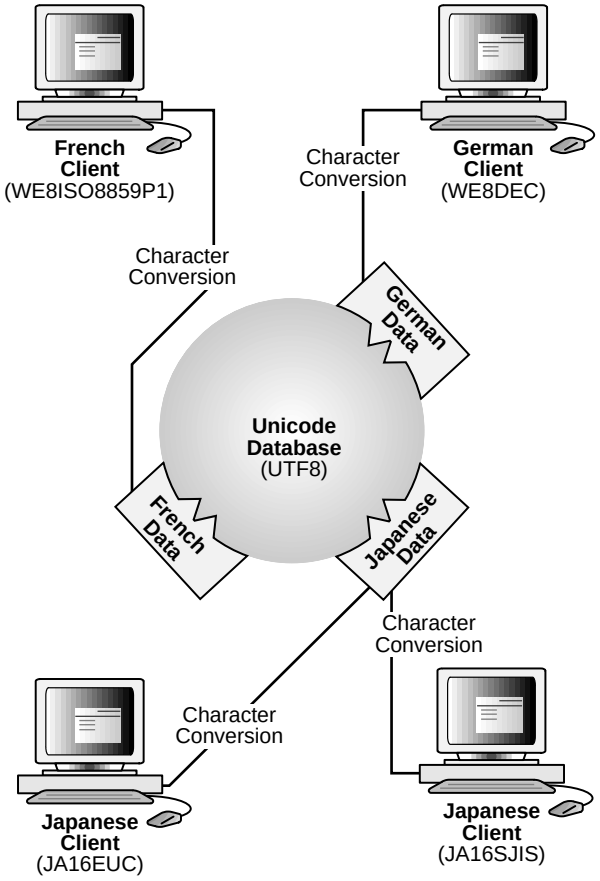
Unrestricted Multilingual Support

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: UCS2 and UTF8. UCS2 is a two-byte fixed-width format; UTF8 is a multi-byte format with a variable width. Oracle8*i* provides support for the UTF8 format. This enhancement is transparent to clients who already provide support for multi-byte character sets.

Character set conversion between a UTF8 database and any single-byte character set introduces very little overhead. Conversion between UTF8 and any multi-byte character set has some overhead but there is no conversion loss problem except that some multi-byte character sets do not support user-defined characters during character set conversion to and from UTF8. See Appendix A, "Locale Data", for further information.

Figure 3–7, "Unrestricted Multilingual Support Example", 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. Please note that each client accesses only data it can process. If Japanese data were retrieved, modified, and stored back by the German client, all Japanese characters would be lost during the character set conversion.

Figure 3–7 Unrestricted Multilingual Support Example



4

SQL Programming

This chapter contains information useful for SQL programming in an NLS environment, including:

- 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:

```
SQL> 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:

```
SQL> 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 multi-byte 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 4–1:

Table 4–1 SQL Functions and Their Parameters

SQL Function	Valid NLS Parameters
TO_DATE	NLS_DATE_LANGUAGE NLS_CALENDAR
TO_NUMBER:	NLS_NUMERIC_CHARACTERS NLS_CURRENCY NLS_DUAL_CURRENCY NLS_ISO_CURRENCY
TO_CHAR	NLS_DATE_LANGUAGE NLS_NUMERIC_CHARACTERS NLS_CURRENCY NLS_ISO_CURRENCY NLS_DUAL_CURRENCY NLS_CALENDAR
NLS_UPPER	NLS_SORT
NLS_LOWER	NLS_SORT

Table 4–1 SQL Functions and Their Parameters

SQL Function	Valid NLS Parameters
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, 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

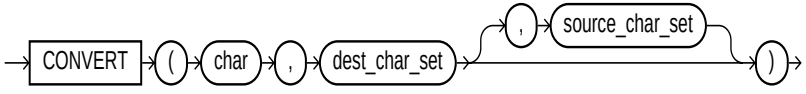
The SQL function CONVERT allows for conversion of character data between character sets.

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.

If the CONVERT function is used in a stored procedure, the stored procedure runs independently of the client character set (that is, it uses the server's character set), which sometimes results in the last converted character being truncated.

The syntax for CONVERT is:

Figure 4–1 CONVERT Syntax



where *source_char_set* is the source character set and *dest_char_set* is the destination 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.

For more information on CONVERT, see *Oracle8i SQL Reference*.

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.

The NLS_CHARSET_DECL_LEN function returns the declaration length (in number of characters) for an NCHAR column.

For more information on these functions, see *Oracle8i 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*, the function returns the ID of the server's database character set. If the value NCHAR_CS is entered for *TEXT*, 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.

```
SQL> 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.

```
SQL> 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 two methods.

1. Use of the NLSSORT() function in the WHERE clause.

   ```
   SQL> SELECT *
      > FROM table1
      > WHERE NLSSORT(col1, 'NLS_SORT = FRENCH')>
   > NLSSORT(col2, 'NLS_SORT = FRENCH');
   ```

2. Setting the session parameter NLS_COMP to ANSI, in which case the NLS_SORT session parameter is used in the WHERE clause.

   ```
   SQL> 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:

```
SQL> SELECT * FROM emps
   > 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 8859/1 8-bit character set, the following query:

```
SQL> SELECT col1 FROM tab1 WHERE col1 > 'B';
```

returns both BCD and ÄBC because Ä has a higher numeric value than B. However, in German, an Ä is sorted alphabetically before B. Such conventions are language dependent even when the same character is used. In Swedish, an 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 is binary. To use linguistic comparison, the NLSSORT function can be used. Sometimes this can be tedious, especially when the linguistic sort needed has already been specified in the NLS_SORT session parameter. One 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:

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

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

```
SQL> 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:

```
SQL> 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')
```

Pattern Matching Characters for Fixed-Width Multi-Byte Character Sets

The LIKE operator is used in character string comparisons with pattern matching. Its syntax requires the use of two special pattern matching characters: the underscore (_) and the percent sign(%). The space character is used to pad CHAR values to the declared column length.

When the LIKE operator is applied to a national character set column (NCHAR or NVARCHAR2) or a value of an NCHAR column must be padded to its declared length, and the national character set is fixed-byte multi-width, a problem arises because the (single-byte) underscore, percent, and space characters are not present in the character set. Table 4–2 lists characters that should be used instead.

Table 4–2 Encoding for the Underscore, Percent Sign, and Pad Character

For This Character Set	Use These Code Point Values		
	Underscore	**Percent Sign**	**Pad Character (Space)**
JA16SJISFIXED	0x8151	0x8193	0x8140
JA16EUCFIXED	0xa1b2	0xa1f3	0xa1a1
JA16DBCSFIXED	0x426d	0x426c	0x4040
ZHT32TRISFIXED	0x8eb1a1df	0x8eb1a1a5	0x8ebla1a0

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-ijan1)/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.

For complete information on using date and number masks, see *Oracle8i SQL Reference*.

Miscellaneous Topics

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.

5

OCI Programming

This chapter contains information useful for OCI programming, including:

- Using the OCI NLS Functions
- NLS Language Information Retrieval
- String Manipulation
- Character Classification
- Character Set Conversion
- Messaging Mechanism

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.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE, or OCI_ERROR on wrong items.

Table 5–1 OCINlsGetInfo Keywords/Parameters

Keyword/ Parameter	Meaning
hndl(IN / OUT)	The OCI environment or user session handle initialized in object mode
errhp(IN / OUT)	The OCI error handle. If there is an error, it is recorded in errhp and this function returns a NULL pointer. Diagnostic information can be obtained by calling OCIErrorGet()
buf(OUT)	Pointer to the destination buffer

Table 5–1 OCINlsGetInfo Keywords/Parameters

Keyword/ Parameter	Meaning
buflen(IN)	The size of the destination buffer. The maximum length for each piece of information is OCI_NLS_MAXBUFSZ bytes
item(IN)	Specifies which item in OCI environment handle to return. Can be one of the following values: OCI_NLS_DAYNAME1: Native name for Monday OCI_NLS_DAYNAME2: Native name for Tuesday OCI_NLS_DAYNAME3: Native name for Wednesday OCI_NLS_DAYNAME4: Native name for Thursday OCI_NLS_DAYNAME5: Native name for Friday OCI_NLS_DAYNAME6: Native name for Saturday OCI_NLS_DAYNAME7: Native name for Sunday OCI_NLS_ABDAYNAME1: Native abbreviated name for Monday OCI_NLS_ABDAYNAME2: Native abbreviated name for Tuesday OCI_NLS_ABDAYNAME3: Native abbreviated name for Wednesday OCI_NLS_ABDAYNAME4: Native abbreviated name for Thursday OCI_NLS_ABDAYNAME5: Native abbreviated name for Friday OCI_NLS_ABDAYNAME6: Native abbreviated name for Saturday OCI_NLS_ABDAYNAME7: Native abbreviated name for Sunday

Table 5–1 OCINlsGetInfo Keywords/Parameters

Keyword/ Parameter	Meaning
	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 5–1 OCINlsGetInfo Keywords/Parameters

Keyword/ Parameter	Meaning
	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

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 *Oracle Call Interface Programmer's Guide* and *Oracle8i Data Cartridge Developer's Guide* for further information.

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

Two types of data structure are supported for string manipulation: multi-byte string and wide character string. Multi-byte 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 data type 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 implementation- and platform-dependent. 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 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 compare, concatenation and searching

Table 5–2 OCI String Manipulation Calls

Function Call	Description
OCIMultiByteToWideChar()	Converts an entire null-terminated string into the wchar format
OCIMultiByteInSizeTo WideChar()	Converts part of a string into the wchar format
OCIWideCharToMultiByte()	Converts an entire null-terminated wide character string into a multi-byte string
OCIWideCharInSizeTo MultiByte()	Converts part of a wide character string into the multi-byte format
OCIWideCharToLower()	If there is a lower-case character mapping in the specified locale, it will return the lower-case in wide character. If not, returns the same wide character
OCIWideCharToUpper()	If there is an lower-case character mapping in the specified locale, it will return the upper-case in wide character. If not, returns the same wide character
OCIWideCharStrcmp()	Compares two wide character strings in binary, linguistic, or case-insensitive manners
OCIWideCharStrncmp()	Similar to OCIWideCharStrcmp(), but compares two multi-byte strings in binary, linguistic, or case-insensitive manners, except that at most len1 bytes form str1 and len2 bytes form str2 are compared
OCIWideCharStrcat()	Appends a copy of the string pointed to by wsrcstr. Then returns the number of characters in the resulting string
OCIWideCharStrchr()	Searches for the first occurrence of wc in the string pointed to by wstr. Then returns a pointer to the wchar if successful
OCIWideCharStrcpy()	Copies the wchar string pointed to by wsrcstr into the array pointed to by wdststr. Then returns the number of characters copied
OCIWideCharStrlen()	Computes the number of characters in the wchar string pointed to by wstr, and returns this number

Table 5–2 OCI String Manipulation Calls

Function Call	Description
OCIWideCharStrncat()	Appends a copy of the string pointed to by wsrcstr. Then returns the number of characters in the resulting string, except that at most n characters are appended
OCIWideCharStrncpy()	Copies the wchar string pointed to by wsrcstr into the array pointed to by wdststr. Then returns the number of characters copied, except that at most n characters are copied from the array
OCIWideCharStrrchr()	Searches for the last occurrence of wc in the string pointed to by wstr
OCIWideCharStrCase Conversion()	Converts the wide character string pointed to by wsrcstr into case specified by flag and copies the result into the array pointed to by wdststr
OCIWideCharDisplayLength()	Determines the number of column positions required for wc in display
OCIWideCharMultibyte Length()	Determines the number of bytes required for wc in multi-byte encoding
OCIMultiByteStrcmp()	Compares two multi-byte strings in binary, linguistic, or case-insensitive manners
OCIMultiByteStrncmp()	Compares two multi-byte strings in binary, linguistic, or case-insensitive manners, except that at most len1 bytes form str1 and len2 bytes form str2 are compared
OCIMultiByteStrcat()	Appends a copy of the multi-byte string pointed to by srcstr
OCIMultiByteStrcpy()	Copies the multi-byte string pointed to by srcstr into an array pointed to by dststr. It returns the number of bytes copied
OCIMultiByteStrlen()	Computes the number of bytes in the multi-byte string pointed to by str, and returns this number
OCIMultiByteStrncat()	Appends a copy of the multi-byte string pointed to by srcstr, except that at most n bytes from srcstr are appended to dststr
OCIMultiByteStrncpy()	Copies the multi-byte string pointed to by srcstr into an array pointed to by dststr. It returns the number of bytes copied, except that at most n bytes are copied from the array pointed to by srcstr to the array pointed to by dststr
OCIMultiByteStrnDisplay Length()	Returns the number of display positions occupied by the complete characters within the range of n bytes
OCIMultiByteStrCase Conversion()	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 5–3 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, 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 5–4 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

Table 5–4 OCIMultiByteInSizeToWideChar Keywords/Parameters

Keyword/Parameter	Meaning
dstsz(IN)	Destination buffer size in character. If it is zero, this function just returns number of characters will be need for the conversion
src (IN)	Source string to be converted
srcsz(IN)	Length of source string in byte
rsize(OUT)	Number of characters written into destination buffer, or number of characters for converted string is dstsz is zero. If it is a NULL pointer, nothing to return

OCIWideCharToMultiByte

Syntax

```
sword OCIWideCharToMultiByte(dvoid *hndl, OraText *dst, CONST OCIWchar *src, size_t *rsize)
```

Remarks

This routine converts an entire NULL-terminated wide character string into a multi-byte string. The output buffer will be NULL-terminated.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR.

Table 5–5 OCIWideCharToMultiByte 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 multi-byte string
src(IN)	Source wchar string to be converted
srcsz(IN)	Length of source string in byte
rsize(OUT)	Number of characters written into destination buffer. If it is a NULL pointer, nothing will be returned

OCIWideCharInSizeToMultiByte

```
sword OCIWideCharInSizeToMultiByte(dvoid *hndl, OraText *dst, size_t dstsz,
CONST OCIWchar *src, size_t srcsz, size_t *rsize)
```

Remarks

This routine converts part of wchar string into the multi-byte 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 ending NULL-terminator needed to store the converted string.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR.

Table 5–6 OCIWideCharInSizeToMultiByte 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 multi-byte. It can be a NULL pointer if dstsz is zero
dstsz(IN)	Destination buffer size in byte. If it is zero, it just returns the size of bytes need for converted string
src(IN)	Source wchar string to be converted
srcsz(IN)	Length of source string in character
rsize(OUT)	Number of bytes written into destination buffer, or number of bytes need to store the converted string if dstsz is zero. If it is a NULL pointer, nothing to return

OCIWideCharToLower

Syntax

```
OCIWchar OCIWideCharToLower(dvoid *hndl, OCIWchar wc)
```

Remarks

If there is a lower-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 5–7 OCIWideCharToLower Keywords/Parameters

Keyword/Parameter	Meaning
hndl(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 *hndl, OCIWchar wc)
```

Remarks

If there is a upper-case character mapping for wc in the specified locale, it will return the upper-case in wchar, it will return wc itself otherwise.

Returns

A wchar.

Table 5–8 OCIWideCharToUpper Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar for upper-case mapping

OCIWideCharStrcmp

Syntax

```
int OCIWideCharStrcmp(dvoid *hndl, 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 5–9 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 default value. OCI_NLS_LINGUISTIC: for linguistic comparison specified in the locale. This flag can be ORed 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 5–10 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 5–10 (Cont.) OCIWideCharStrncmp Keywords/Parameters

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 linguistic comparison specified in the locale. This flag can be ORed 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 5–11 OCIWideCharStrcat Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN /)	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 5–12 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 wchar string to search
wc(IN)	wchar to search for

OCIWideCharStrcpy

Syntax

```
size_t OCIWideCharStrcpy(dvoid *hndl, OCIWchar *wdststr, CONST OCIWchar *wsrcstr)
```

Remarks

This function copies the wchar string pointed to by wsrcstr, including the NULL-terminator, into the array pointed to by wdststr.

Returns

The number of characters copied not including the ending NULL-terminator.

Table 5–13 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 wchar buffer
wsrcstr(IN)	Pointer to the source wchar 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 5–14 OCIWideCharStrlen Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wstr(IN)	Pointer to the source wchar string

OCIWideCharStrncat

Syntax

```
size_t OCIWideCharStrncat(dvoid *hndl, OCIWchar *wdststr, CONST OCIWchar *wsrcstr, size_t n)
```

Remarks

This function is similar to OCIWideCharStrcat(), except that at most n characters from wsrcstr are appended to wdststr. Note that the NULL-terminator in wsrcstr will stop appending. wdststr will be NULL-terminated.

Returns

The number of characters in the result string, not including the ending NULL-terminator.

Table 5–15 OCIWideCharStrncat 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

Table 5–15 OCIWideCharStrncat Keywords/Parameters

Keyword/Parameter	Meaning
n(IN)	Number of characters from wsrcstr 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 5–16 OCIWideCharStrncpy 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 wchar buffer
wsrcstr(IN)	Pointer to the source wchar string
n(IN)	Number of characters from wsrcstr 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 5–17 OCIWideCharStrrchr Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wstr(IN)	Pointer to the wchar string to search
wc(IN)	wchar to search for

OCIWideCharStrCaseConversion

Syntax

```
size_t OCIWideCharStrCaseConversion(dvoid *hndl, OCIWchar *wdststr, CONST OCIWchar*wsrcstr, ub4
flag)
```

Remarks

This function converts the wide char string pointed to by wsrcstr into the uppercase or lowercase specified by 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 result string not including NULL-terminator.

Table 5–18 OCIWideCharStrCaseConversion Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle
wdststr(OUT)	Pointer to destination array
wsrcstr(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 ORed 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 5–19 OCIWideCharDisplayLength Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar character

OCIWideCharMultiByteLength

Syntax

```
size_t OCIWideCharMultiByteLen(dvoid *hndl, OCIWchar wc)
```

Remarks

This function determines the number of byte required for wc in multi-byte encoding. It returns the number of bytes in multi-byte for wc.

Returns

The number of bytes.

Table 5–20 OCIWideCharMultiByteLength Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar character

OCIMultiByteStrcmp

Syntax

```
int OCIMultiByteStrcmp(dvoid *hndl, CONST OraText *str1, CONST OraText *str2, int flag)
```

Remarks

It compares two multi-byte 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 5–21 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)	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 linguistic comparison specified in the locale. This flag can be ORed with OCI_NLS_CASE_INSENSITIVE for case-insensitive comparison

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 5–22 OCIMultiByteStrncmp Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle
str1(IN)	Pointer to the first string
len1(IN)	The length for the first string for comparison
str2(IN)	Pointer to the second 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 linguistic comparison specified in the locale. This flag can be ORed 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 multi-byte 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 5–23 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 multi-byte 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 multi-byte 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 5–24 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 multi-byte string

OCIMultiByteStrlen

Syntax

```
size_t OCIMultiByteStrlen(dvoid *hndl, CONST OraText *str)
```

Remarks

This function computes the number of bytes in the multi-byte 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 5–25 OCIMultiByteStrlen Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	Pointer to the OCI environment or user session handle
str(IN)	Pointer to the source multi-byte 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 5–26 OCIMultiByteStrncat Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	Pointer to OCI environment or user session handle
dststr(IN / OUT)	Pointer to the destination multi-byte string for appending
srcstr(IN)	Pointer to the source multi-byte string to append
n(IN)	The number of bytes from srcstr to append

OCIMultiByteStrncpy

Syntax

```
size_t OCIMultiByteStrncpy(dvoid *hndl, OraText *dststr, CONST OraText *srcstr, size_t n)
```

Remarks

This function is similar to OCIMultiByteStrcpy(), 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 5–27 OCIMultiByteStrncpy Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	Pointer to OCI environment or user session handle
srcstr(OUT)	Pointer to the destination buffer
dststr(IN)	Pointer to the source multi-byte string
n(IN)	The number of bytes from srcstr 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 5–28 OCIMultiByteStrncpy Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle
str(IN)	Pointer to a multi-byte string
n(IN)	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 multi-byte 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 5–29 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 ORed 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 *Oracle Call Interface Programmer's Guide* and *Oracle8i Data Cartridge Developer's Guide* for further information.

Character Classification

The Oracle Call Interface offers many function calls for classifying characters.

Table 5–30 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
OCIWideCharIsUpper()	Tests whether the wide character is an uppercase character
OCIWideCharIsXdigit()	Tests whether the wide character is a hexadecimal digit
OCIWideCharIsSingleByte()	Tests whether wc is a single-byte character when converted into multi-byte

OCIWideCharIsAlnum

Syntax

```
boolean OCIWideCharIsAlnum(dvoid *hndl, OCIWchar wc)
```

Remarks
It tests whether wc is a letter or decimal digit.

Returns
TRUE or FALSE.

Table 5–31 OCIWideCharIsAlnum Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	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 5–32 OCIWideCharIsAlpha Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar for testing

OCIWideCharIsCntrl

Syntax

```
boolean OCIWideCharIsCntrl(dvoid *hndl, OCIWchar wc)
```

Remarks
It tests whether wc is a control character.

Returns
TRUE or FALSE.

Table 5–33 OCIWideCharIsCntrl Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar for testing

OCIWideCharIsDigit

Syntax

```
boolean OCIWideCharIsDigit(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether wc is a decimal digit character.

Returns

TRUE or FALSE.

Table 5–34 OCIWideCharIsDigit Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar 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 alphabetic letter, decimal digit, and punctuation.

Returns

TRUE or FALSE.

Table 5–35 OCIWideCharIsGraph Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar for testing

OCIWideCharIsLower

Syntax

```
boolean OCIWideCharIsLower(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether wc is a lowercase letter.

Returns

TRUE or FALSE.

Table 5–36 OCIWideCharIsLower Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar for testing

OCIWideCharIsPrint

Syntax

```
boolean OCIWideCharIsPrint(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether wc is a printable character.

Returns

TRUE or FALSE.

Table 5–37 OCIWideCharIsPrint Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set

Table 5–37 OCIWideCharIsPrint Keywords/Parameters

Keyword/Parameter	Meaning
wc(IN)	wchar for testing

OCIWideCharIsPunct

Syntax

```
boolean OCIWideCharIsPunct(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether wc is a punctuation character.

Returns

TRUE or FALSE.

Table 5–38 OCIWideCharIsPunct Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar 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 5–39 OCIWideCharIsSpace Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set

Table 5–39 OCIWideCharIsSpace Keywords/Parameters

Keyword/Parameter	Meaning
wc(IN)	wchar for testing

OCIWideCharIsUpper

Syntax

```
boolean OCIWideCharIsUpper(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether wc is an uppercase letter.

Returns

TRUE or FALSE.

Table 5–40 OCIWideCharIsUpper Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar 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 5–41 OCIWideCharIsXdigit Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar for testing

OCIWideCharIsSingleByte

Syntax

```
boolean OCIWideCharIsSingleByte(dvoid *hndl, OCIWchar wc)
```

Remarks

It tests whether wc is a single-byte character when converted into multi-byte.

Returns

TRUE or FALSE.

Table 5–42 OCIWideCharIsSingleByte Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	OCI environment or user session handle to determine the character set
wc(IN)	wchar for testing

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 *Oracle Call Interface Programmer's Guide* and *Oracle8i Data Cartridge Developer's Guide* for further information.

Character Set Conversion

Conversion between Oracle character set and Unicode (16 bit, fixed width Unicode encoding) is supported. Replacement characters will be used if there is no mapping from Unicode to the Oracle character set, therefore, round-trip conversion is not always possible.

Table 5–43 OCI Character Set Conversion Calls

Function Call	Description
OCICharsetToUnicode()	Converts a multi-byte string pointed to by src to Unicode into the array pointed to by dst
OCIUnicodeToCharset()	Converts a Unicode string pointed to by src to multi-byte into the array pointed to by dst
OCICharSetConversionIs ReplacementUsed()	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 multi-byte string pointed to by src to Unicode 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 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 5–44 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 character
src(IN)	Pointer to multi-byte 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 multi-byte 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 multi-byte. 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 5–45 OCIUnicodeToCharSet 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 destination buffer in bytes
src(IN)	Pointer to a Unicode string

Table 5–45 OCIUnicodeToCharSet Keywords/Parameters

Keyword/Parameter	Meaning
srclen(IN)	The size of source string in characters
rsize(OUT)	The number of bytes converted. If it is a NULL pointer, nothing to return

OCICharSetConversionIsReplacementUsed

Syntax

```
boolean OCICharSetConversionIsReplacementUsed(dvoid *hndl)
```

Remarks

This function indicates whether or not the replacement character was used for nonconvertible characters in character set conversion in the last invocation of OCICharSetToUnicode().

Returns

TRUE is the replacement character was used in last OCICharsetConv() invoking, else FALSE.

Table 5–46 OCICharSetConversionIsReplacementUsed Keywords/Parameters

Keyword/Parameter	Meaning
hndl(IN / OUT)	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 will be used if there is no mapping from Unicode to the Oracle character set, thus, round-trip conversion is not always possible.

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 *Oracle Call Interface Programmer's Guide* and *Oracle8i Data Cartridge Developer's Guide* for further information.

Messaging Mechanism

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

Table 5–47 OCI Messaging Function Calls

Function Call	Description
OCIMessageOpen()	Opens a message handle for facility of product in a language pointed to by hndl
OCIMessageGet()	Retrieves a message with message number identified by msgno and if the buffer is not zero, the function will copy the message into the buffer pointed to by msgbuf
OCIMessageClose()	Closes a message handle pointed to by msgh and frees any memory associated with this handle

See *Oracle Call Interface Programmer's Guide* and *Oracle8i Data Cartridge Developer's Guide* for further information.

OCIMessageOpen

Syntax

```
sword OCIMessageOpen(dvoid *hndl, OCIError *errhp, OCIMsg **msghp, CONST OraText *product, CONST
OraText *facility, OCIDuration dur)
```

Remarks

This function opens a message handle for facility of product 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, else it will use the default message file which is for American language for the facility. The function returns a pointer pointed to a message handle into the msghp parameter.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR.

Table 5–48 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 record 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 message in a system dependent way. For example, in Solaris, the directory of message files for the product 'rdbms' 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 facility 'img' in the American language will be 'imgus.msb' where 'us' is the abbreviation for the American language and 'msb' as message binary file extension

Table 5–48 OCICharSetConversionIsReplacementUsed Keywords/Parameters

Keyword/Parameter	Meaning
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 *msgh, ub4 msgno, OraText *msgbuf, size_t buflen)
```

Remarks

This function will get message with message number identified by msgno and if buflen is not zero, the function will copy the message into the buffer pointed to by msgbuf. If buflen is zero, the message will be copied into a message buffer inside the message handle pointed to by msgh. For both cases. it will return the pointer to the NULL-terminated message string. If it cannot get the message required, it will return a NULL pointer.

Returns

A pointer to a NULL-terminated message string on success, otherwise a NULL pointer.

Table 5–49 OCIMessageGet Keywords/Parameters

Keyword/Parameter	Meaning
msgh(IN / OUT)	Pointer to a message handle which was previously opened by OCIMessageOpen()
msgno(IN)	The message number for getting message
msgbuf(OUT)	Pointer to a destination buffer to the message retrieved. If buflen is zero, it can be NULL pointer
buflen(IN)	The size of the above destination buffer

OCIMessageClose

Syntax

```
sword OCIMessageClose(dvoid *hndl, OCIError *errhp, OCIMsg *msgh)
```

Remarks

This function closes a message handle pointed to by msgh and frees any memory associated with this handle.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR.

Table 5–50 OCIMessageClose 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 record in errhp and this function returns a NULL pointer. Diagnostic information can be obtained by calling OCIErrorGet()
msgh(IN / OUT)	A pointer to a message handle that was previously opened by OCIMessageOpen()

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>   the name of the product
   <facility>  the name of the facility
    [language]  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 start with "/" and "/ /" are treated as internal comments and hence 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 used currently, simply 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 will retrieve 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 message file **$HOME/myApp/mesg/impus.msb** will be used.

Message file

Lmsgen will convert the message file (impus.msg) into binary format (impus.msb).

The following is a portion of the text message file, impus.msg:

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

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);
```

6

Java

This chapter covers NLS issues with the use of Java. It contains:

Overview of Oracle8*i* Java Support

Java support is included in all tiers of a multi-tier computing environment in order to enable users to develop and deploy Java programs. You can run Java classes as Java stored procedures, Java CORBA objects, and Enterprise Java Beans (EJB) on the Java Virtual Machine (VM) of the database server. Users are able to develop a Java class, load it into the database, and package it as a stored procedure callable from SQL. Users can also develop a standard Java CORBA object or EJB, load the related classes into the database and publish them as a named object callable from any CORBA or EJB client.

The JDBC driver and SQLJ translator are also provided as programmatic interfaces enabling Java programs to access the Oracle8*i* database. Users 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 Oracle8*i*.

This chapter examines the NLS support for individual Java components. Typical database and client configurations are discussed for multilingual application deployment, and how the Java components are used in the midst of them. Finally, the design and implementation of a demo application are explored to demonstrate how Oracle's Java support is used to make the application run in a multilingual environment.

Java components provide NLS support and use Unicode as the multilingual character set of choice. The following are Oracle8*i*'s Java components:

- *JDBC Driver* - Oracle provides JDBC as the core programmatic interface for accessing Oracle8*i* 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 JDBC Server 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.

- *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 1.0 container is built into the database server to provide a platform to develop and deploy EJBs.

JDBC

This section describes the following:

Oracle JDBC drivers provide globalization support by allowing users to retrieve data from or insert data into a database in any character set that Oracle supports. Because Java strings are UCS2 encoded (16-bit Unicode) for JDBC programs, the target character set on the client is always UCS2. Character set conversion is required to convert data from the database character set (Db Charset) to UCS2. This applies to CHAR, LONG, CLOB, and VARCHAR2 data types; RAW data is not converted.

Following are a few examples of commonly used Java methods for JDBC that rely heavily on NLS character set conversion:

- `java.sql.ResultSet`'s methods `getString()` and `getUnicodeStream()` return values from the database as Java strings and as a stream of Unicode characters, respectively.
- `oracle.sql.CLOB`'s method `getCharacterStream()` returns the contents of a `CLOB` as a Unicode stream.
- `oracle.sql.CHAR`'s methods `getString()`, `toString()`, and `getStringWithReplacement()`.

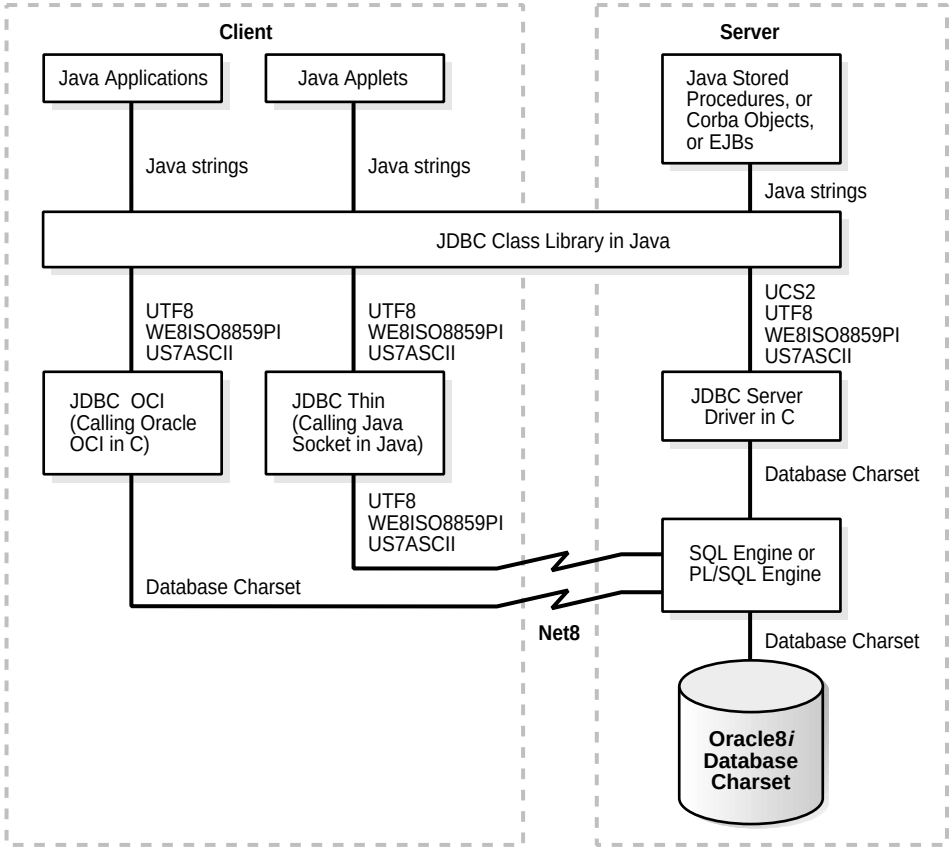
The techniques that Oracle's drivers use to perform character set conversion for Java applications depend on the character set the database uses. The simplest case is where the database uses a US7ASCII or WE8ISO8859P1 character set. In this case, the driver converts the data directly from the database character set to UCS2,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 UCS2.

Note: The JDBC drivers perform all character set conversions transparently. No user intervention is necessary for the conversions to occur.

Figure 6–1 presents a graphical view of how data is converted in JDBC drivers.

Figure 6–1 Data Conversion in JDBC Drivers



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 UCS2 for the JDBC server-side driver. The JDBC Class Library converts the input stream to UCS2 before passing it to the client applications. If the input stream is in UTF8, the JDBC Class Library converts the UTF8 encoded string to UCS2 by using the bit-wise operation defined in the UTF8-to-UCS2 conversion algorithm. If the input stream is in US7ASCII or WE8ISO8859P1, it converts the input string to UCS2 by casting the bytes to Java

characters. This is based on the first 128 and 256 characters of UCS2 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.

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.

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 time 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 UCS2 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 UCS2.

Notes:

- The driver sets the value of NLS_LANG to UTF8 to minimize the number of conversions it performs in Java. It performs the conversion from the database character set to UTF8 in C.
- The change to UTF8 is for the JDBC application process only.
- For more information on the NLS_LANG parameter, see "Choosing a Locale with NLS_LANG" on page 2-4.

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 UCS2 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 UCS2 in Java.

Note: The OCI and Thin drivers both provide the same transparent support for NLS.

JDBC Server Driver

For Java classes running in the Java VM of the Oracle8*i* Server, the JDBC Server driver is used to talk to the SQL engine or the PL/SQL engine for SQL processing. Because the JDBC Server 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 driver, and the data is passed to or from the JDBC Class Library as is. Otherwise, the JDBC Server driver converts the data from the database character set to UCS2 before passing them to and from the class library. The class library does not need to do any conversion in this case.

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.

JDBC constructs and populates `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.

Additionally, you might want to construct an `oracle.sql.CHAR` object yourself (to pass into a prepared statement, for example). When you construct an `oracle.sql.CHAR` object, you must provide character set information to the `oracle.sql.CHAR` object by way of an instance of the `oracle.sql.CharacterSet` class. Each instance of the `CharacterSet` class represents one of the NLS character sets that Oracle supports.

Follow these general steps to construct an `oracle.sql.CHAR` object:

1. Create a `CharacterSet` instance by calling the static `CharacterSet.make()` method. This method is a factory for the character set class. It takes as input an integer `OracleId`, which corresponds to a character set that Oracle supports. For example:

```
int OracleId = CharacterSet.JA16SJIS_CHARSET; // this is character set 832
...
CharacterSet mycharset = CharacterSet.make(OracleId);
```

Each character set that Oracle supports has a unique predefined `OracleId`. The `OracleId` can always be referenced as a character set <Oracle charset name>_CHARSET where <Oracle charset name> is the Oracle character set.

2. Construct an `oracle.sql.CHAR` object. Pass to the constructor a string (or the bytes that represent the string) and the `CharacterSet` object that indicates how to interpret the bytes based on the character set. For example:

```
String mystring = "teststring";
...
oracle.sql.CHAR mychar = new oracle.sql.CHAR(teststring, mycharset);
```

   The `oracle.sql.CHAR` class has multiple constructors: they can take a string, a byte array, or an object as input along with the `CharacterSet` object. In the case of a string, the string is converted to the character set indicated by the `CharacterSet` object before being placed into the `oracle.sql.CHAR` object.

   Refer to the `oracle.sql.CHAR` class Javadoc for more information.

The server (database) and the client (or application running on the client) can use different character sets. When you use the methods of this class to transfer data between the server and the client, the JDBC drivers must convert the data from the server character set to the client character set (or vice versa).

The `oracle.sql.CHAR` in Oracle Object Types

In Oracle8*i*, 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 6–1, "Data Conversion in JDBC Drivers", 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. An example of an object type created by 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 UCS2 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

Data Size Restriction for NLS Conversions

There is a limit on the maximum sizes for CHAR and VARCHAR2 datatypes when used in bind calls. This limitation is necessary to avoid data corruption. Data corruption occurs only with binds (not for defines) and it affects only CHAR and VARCHAR2 datatypes if you are connected to a multibyte character set database.

The maximum bind lengths are limited in the following way:

CHARs and VARCHAR2s experience character set conversions that can result in an increase in the length of the data in bytes. The ratio between data sizes before and after a conversion is called the NLS Ratio. After conversion, the bind values should not be greater than 4 Kbytes.

Table 6–1 New Restricted Maximum Bind Length for Client-Side Drivers

Driver	Datatype	Old Max Bind Length (bytes)	New Restricted Max Bind Length (bytes)
Thin and OCI	CHAR	2000	min(2000,4000 / NLS_Ratio)
	VARCHAR2	4000	(4000 / NLS_Ratio)

For example, when connecting to an Oracle8 server, you cannot bind more than:

- min (2000, 4000 / NLS_RATIO) for CHAR types

OR

- 4000 / NLS_RATIO for VARCHAR2 types

Table 6–2 contains examples of the NLS Ratio and maximum bind values for some common server character sets.

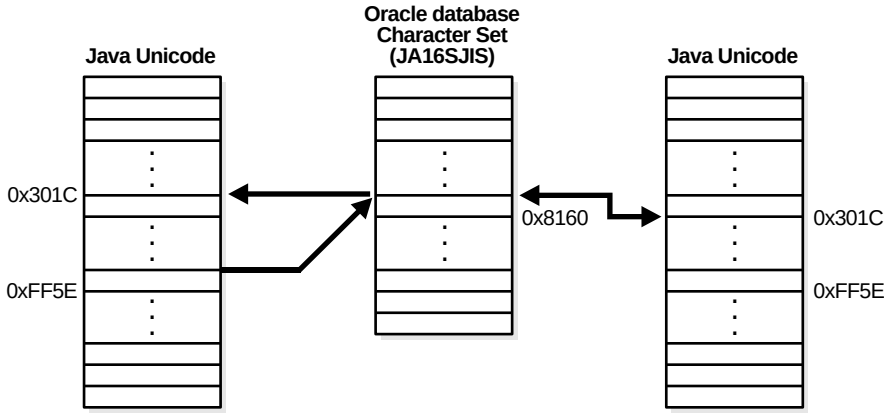
Table 6–2 NLS Ratio and Size Limits for Common Server Character Sets

Server Character Set	NLS Ratio	Maximum Bind Value on Oracle8 Server (in bytes)
WE8DEC	1	4000
US7ASCII	1	4000
ISO 8859-1 through 10	1	4000
JA16SJIS	2	2000
JA16EUC	3	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, i.e., 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 6–2 Character Integrity



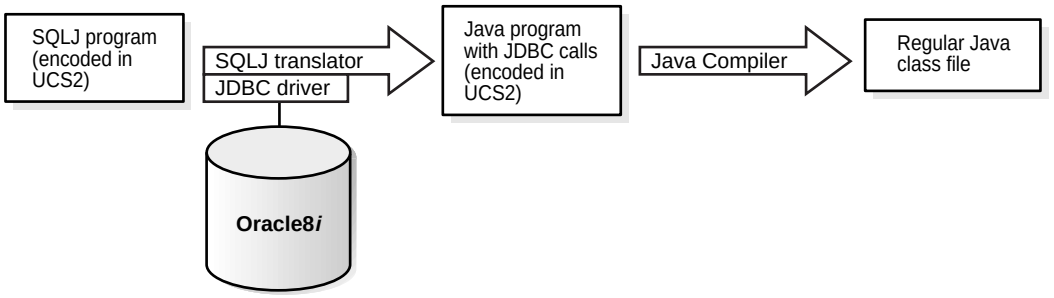
This issue is not a bug in Oracle's JDBC, but rather is an unfortunate side effect of the ambiguity in character mapping specification on different operating systems. Fortunately, this problem affects only a small number of characters in a small number of Oracle character sets such as JA16SJIS, JA16EUC, 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 irrespective of which JDBC driver is used. It also provides a callable interface that the Oracle8*i* 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 6–3 presents a graphical view of how the SQLJ translator works.

Figure 6–3 Using the SQLJ Translator



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 UCS2 encoded SQLJ program is being passed to the SQLJ translator and the Java program output is also encoded in UCS2. 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 UCS2 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;
#sql {insert into E\u0063\u0064 (ENAME, EMPNO) values ('Joe', :empno)};
String name ename = "\ua0a1\ua0a2";
double raise = 0.1;
#sql { update EMP set SAL = :(getNewSal(raise, ename))
where ENAME = :ename;
```

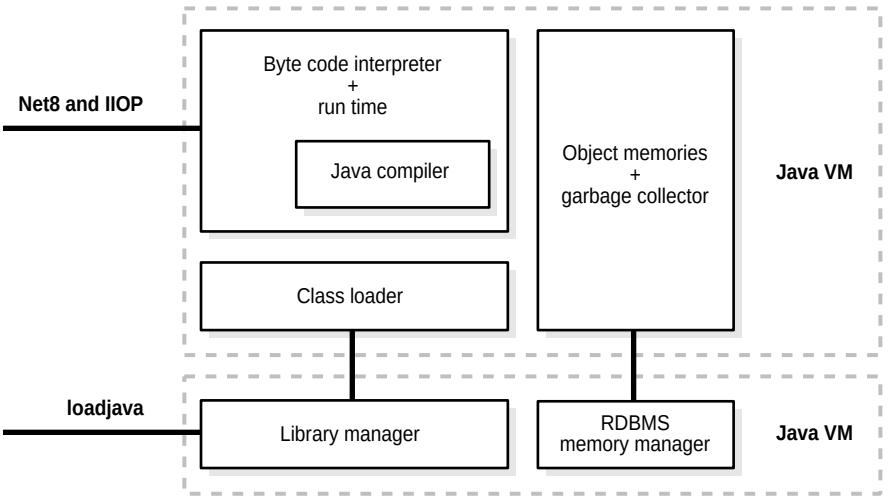
See "Multilingual Demo Applications in SQLJ" on page 6-27 for an example of SQLJ usage for a multilingual Java application.

Java Virtual Machine

The Oracle8*i* Java VM base is integrated into the database server to enable the running of Java classes stored in the database. Oracle8*i* allows user 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 6–4.

Figure 6–4 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. If the class name contains characters that cannot be represented in the database character set, a US7ASCII library unit name is generated and mapped to the real class name stored in a RAW column so that the class loader can find the library unit corresponding to the real class name when the real class name is referenced in a Java program running 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.

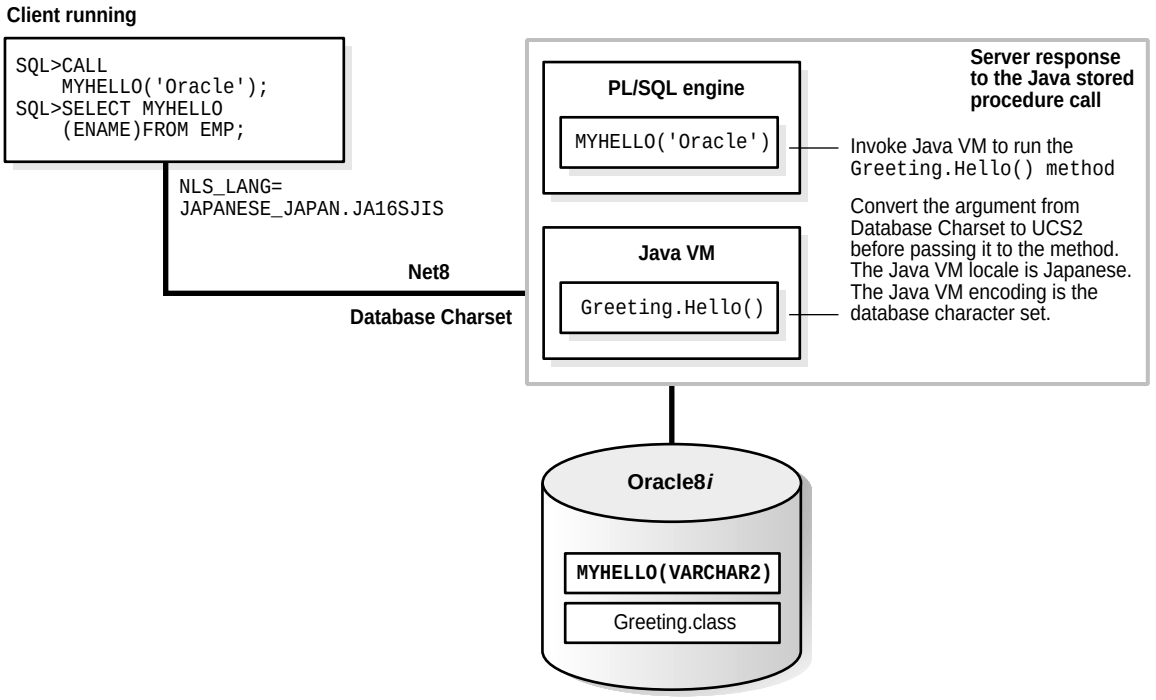
Java Stored Procedures

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 6–5 depicts the runtime environment of a stored function.

Figure 6–5 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; when the PL/SQL engine finds that it is a call specification of the Java method, it calls the Java VM, and 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 UCS2, 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 UCS2-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 on 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.

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. For more information regarding Oracle's CORBA support, see *Oracle8 Database Programming with Java*.

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 to used to resolve a published CORBA object name to the corresponding `PublishedObject`.

CORBA objects implemented in Java in Oracle8*i* 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.

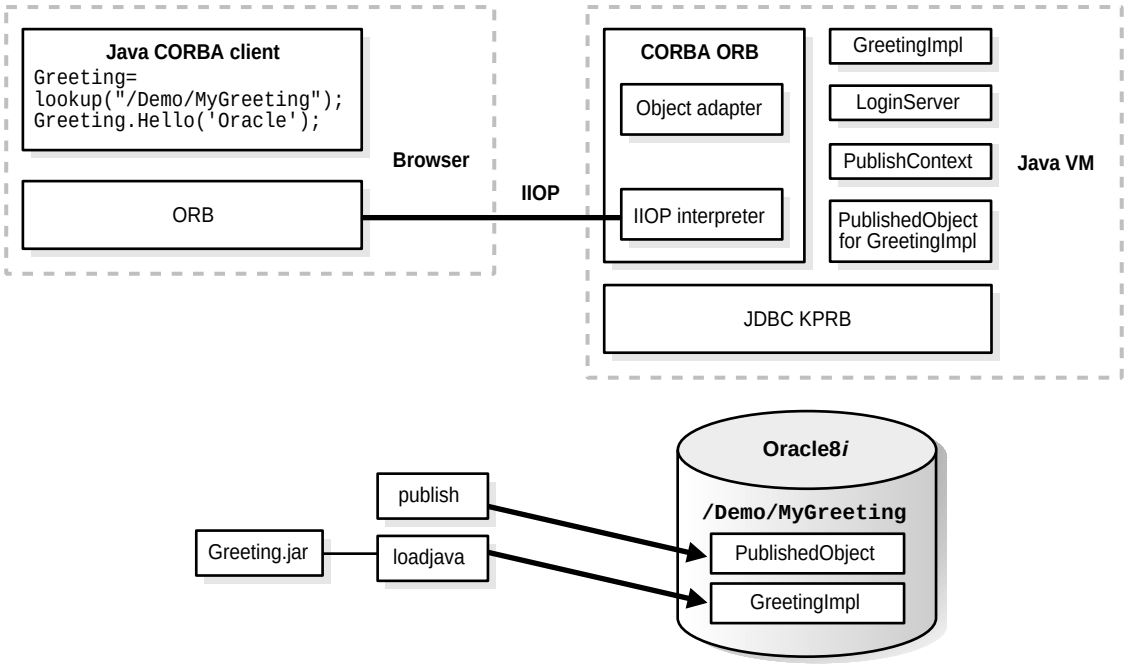
Oracle8*i* implements the CosNaming standard for specifying CORBA object names. CosNaming provides a directory-like structure that is a context for binding names to CORBA objects. A new JNDI URL, "sess_iiop:" is created, and indicates a session based IIOP connection for a CORBA object. A name for a CORBA 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 Oracle8*i* is limited to US7ASCII characters.

Figure 6–6 presents a graphical view of the components in a CORBA environment:

Figure 6–6 Components Supporting CORBA



Java CORBA Object

The CORBA objects for Oracle8*i* 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 _GreetingImplBase
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` data type 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 a excerpt of a client code in Java accessing the `Greeting` object. The ORB is initialized when the CORBA object is first activated via Oracle's implementation of 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_CREDENTIAL);

    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, users can use explicit or implicit authentication:

1. *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.
2. *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 Oracle8*i* 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. For basic concepts about EJBs, refer to *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*. 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*.

1. Use the encoding of the non-US7ASCII character set for the EJB descriptor file and specify -encoding command line argument to tell *ejbdeploy* the encoding of the input file.
2. 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 UCS2 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 (-s) 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 *-encoding* 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. Once 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 8i 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 8i 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, namely the `li.authenticate()` call, 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 Oracle8*i*, the database configurations and client environments for the targeted systems have to be determined first.

Multilingual Database

To choose a database character set for multilingual applications, the following should be considered.

1. *The languages to be supported* - If only single-byte western European languages are to be supported, WE8ISO8859P1 should be used. Otherwise, UTF8 should be used as the database character set. UTF8 can represent data from the Unicode 2.1 standard, which is comprised of characters from the most commonly used languages in the world. UTF8 is good for supporting both single-byte languages and multibyte languages at the same time.

2. *Performance of the database* - If UTF8 is used, Oracle8*i* treats it as a multibyte character set and follows a different code path from that of single-byte character set. In this case, the performance of the UTF8 character set is worse than a single-byte character set such as WE8ISO8859P1, but comparable to a multibyte character set such as JA16SJIS.
3. *Database replication* - If the languages of single- and multibyte character sets are required to be supported and performance cannot be sacrificed, users may consider using UTF8 as a master database and replicate it with a single-byte character set database for European sites, and multibyte character sets for Asian sites. For multilingual sites, the master database will be used.

To use a UTF8 database, consider the following design issues:

1. Because a maximum of 3 bytes is required to represent a Unicode character from any other character set in UTF8, the size of the VARCHAR2 or CHAR data type should be 3 times the number of Unicode characters required to be stored in a column.
2. Because Oracle identifiers use VARCHAR(30) in the data dictionary, the strictest limit for the length of identifier names is 10 Unicode characters in any client character set.

Internationalized Java Server Objects

For each Oracle8*i* session, a separate Java VM instance is created in the server for running the Java object, and Oracle8*i* Java support ensures that the locale of the Java VM instance is the same as that of the client Java VM. 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.

Java objects in the database such as Java stored procedures, Java CORBA, and EJB objects are server objects which are accessible from clients of different language preferences, and therefore, should be internationalized in such a way that they are sensitive to the Java VM locale. For example, with JDK internationalization support, all localizable strings or objects from a Java stored procedure, Java CORBA object, or EJB should be externalized to resource bundles and make the resource bundles as part of the procedure, object, or EJB. With the use of resource bundles, any messages gotten back from the Java server objects will be in the language of the client locale. In addition to the resource bundles, all Java locale-sensitive classes such as date and time formats can be used with the assumption that they will reflect the locale of the calling client.

All Java server objects access the database with the JDBC Server 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 UCS2, and the required conversion from the database character set to UCS2 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 6-20 to ensure Unicode data is being passed from client to server.

Clients of Different Languages

Clients (or middle tiers) can be of different language preferences, database access mechanisms, and Java runtime environments. The following are several commonly used client configurations.

1. **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 log in, 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.
2. **Java applets running in browsers** - Java applets running in browsers can access the Oracle8*i* 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 that of the Java VM running the applets.
3. **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.
4. **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 demo 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 6–7 are defined for storing the book and inventory information of the store.

Figure 6–7 Sample Tables

Book

Field Names	Data Types
ID (PRIMARY KEY)	NUMBER(10)
NAME	VARCHAR(300)
PUBLISH_DATE	DATE
AUTHOR	VARCHAR(120)
PRICES	NUMBER(10,2)

Inventory

Field Names	Data Types
ID (PRIMARY KEY)	NUMBER(10)
LOCATION (PRIMARY KEY)	VARCHAR(90)
QUANTITY	NUMBER(3)

In addition, indexes are built with the NAME and AUTHOR columns of the BOOK table to speed up searching for books. A sequence BOOKSEQ 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 as per 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 since 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");
    }
}
```

Once the methods `Book.remove()` and `Book.add()` 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 become "?". 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 demo program starts up with the user name, password and the location of the database, and 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;
    }
}
```

Summary

Oracle8*i* provides the infrastructure for a multi-tier computing environment to develop and deploy Java applications. JDBC and SQLJ are Java programmatic interfaces for database access used in clients, middle-tiers and servers. The Java VM, CORBA ORB, and EJB container bundled with the Oracle8*i* server provide a run-time environment for Java stored procedures, Java CORBA objects, and EJBs running in the server. Additionally, the CORBA ORB of Oracle8*i* can also be used as a middle tier to other CORBA servers.

The internationalization support in JDBC, SQLJ, Java VM, CORBA ORB, and EJBs and all the supporting services enables the development and deployment of multilingual Java applications for Oracle8*i*. The transparent conversions from a database character set to Unicode enables Java program to manipulate string data from the database in terms of Java strings. The locale of the clients are always preserved in the Java VM of the database server, and that enables the development of locale-sensitive Java stored procedures, Java CORBA objects, and EJBs by leveraging the current internationalization support of the JDK.

A

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 Definitions
- Calendar Systems
- Character Sets that Support the Euro Symbol
- Default Values for NLS Parameters

You can also obtain information about supported character sets, languages, territories, and sorting orders by querying the dynamic data view V$NLS_VALID_VALUES. For more information on the data which can be returned by this view, see *Oracle8i Reference*.

Languages

Table A–1 lists the languages supported by the Oracle server.

Table A–1 Oracle Supported Languages

Name	Abbreviation
AMERICAN	us
ARABIC	ar
BENGALI	bn
BRAZILIAN PORTUGUESE	ptb
BULGARIAN	bg
CANADIAN FRENCH	frc
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
HEBREW	iw
HINDI	hi
HUNGARIAN	hu
ICELANDIC	is
INDONESIAN	in

Table A–1 Oracle Supported Languages

Name	Abbreviation
ITALIAN	i
JAPANESE	ja
KOREAN	ko
LATIN AMERICAN SPANISH	esa
LATVIAN	lv
LITHUANIAN	lt
MALAY	ms
MEXICAN SPANISH	esm
NORWEGIAN	n
POLISH	pl
PORTUGUESE	pt
ROMANIAN	ro
RUSSIAN	ru
SIMPLIFIED CHINESE	zhs
SLOVAK	sk
SLOVENIAN	sl
SPANISH	e
SWEDISH	s
TAMIL	ta
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
LATIN AMERICAN SPANISH	esa
NORWEGIAN	n
POLISH	pl
PORTUGUESE	pt
ROMANIAN	ro
RUSSIAN	ru
SIMPLIFIED CHINESE	zhs
SLOVAK	sk
SPANISH	e

Table A–2 Oracle Supported Messages

Name	Abbreviation
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	ICELAND	QATAR
AMERICA	INDIA	ROMANIA
AUSTRALIA	INDONESIA	SAUDI ARABIA
AUSTRIA	IRAQ	SINGAPORE
BAHRAIN	IRELAND	SLOVAKIA
BANGLADESH	ISRAEL	SLOVENIA
BELGIUM	ITALY	SOMALIA
BRAZIL	JAPAN	SOUTH AFRICA
BULGARIA	JORDAN	SPAIN
CANADA	KAZAKHSTAN	SUDAN
CATALONIA	KOREA	SWEDEN
CHINA	KUWAIT	SWITZERLAND
CIS	LATVIA	SYRIA
CROATIA	LEBANON	TAIWAN
CYPRUS	LIBYA	THAILAND
CZECH REPUBLIC	LITHUANIA	THE NETHERLANDS
DENMARK	LUXEMBOURG	TUNISIA
DJIBOUTI	MALAYSIA	TURKEY
EGYPT	MAURITANIA	UKRAINE

Table A–3 Oracle Supported Territories

Name		
ESTONIA	MEXICO	UNITED ARAB EMIRATES
FINLAND	MOROCCO	UNITED KINGDOM
FRANCE	NEW ZEALAND	UZBEKISTAN
GERMANY	NORWAY	VIETNAM
GREECE	OMAN	YEMEN
HONG KONG	POLAND	
HUNGARY	PORTUGAL	

Character Sets

Oracle-supported character sets are listed below, for easy reference, according to three broad language groups:

- 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 = Multi-byte encoding

FIXED = Fixed-width multi-byte encoding

As mentioned in Chapter 3, "Choosing a Character Set", the type of encoding will affect performance, so you should use the most efficient encoding that meets your language needs. Also, some encoding types can only be used with certain data types. For instance, fixed-width multibyte encoded character sets 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
ZHS16CGB231280	CGB2312-80 16-bit Simplified Chinese	MB, ASCII
JA16EUC	EUC 24-bit Japanese	MB, ASCII
JA16EUCYEN	EUC 24-bit Japanese with '\' mapped to the Japanese yen character	MB
JA16EUCFIXED	EUC 16-bit Japanese. A fixed-width subset of JA16EUC (contains only the 2-byte characters of JA16EUC). Contains no 7- or 8-bit ASCII characters	FIXED
ZHT32EUC	EUC 32-bit Traditional Chinese	MB, ASCII
ZHT32EUCFIXED	EUC 32-bit Traditional Chinese (32-bit fixed-width, no single byte)	FIXED
ZHS16GBK	GBK 16-bit Simplified Chinese	MB, ASCII, UDC
ZHS16GBKFIXED	GBK 16-bit Simplified Chinese (16-bit fixed-width, no single byte)	FIXED, UDC
ZHT16CCDC	HP CCDC 16-bit Traditional Chinese	MB, ASCII
JA16DBCS	IBM EBCDIC 16-bit Japanese	MB, UDC

Table A–4 Asian Language Character Sets

Name	Description	Comments
JA16EBCDIC930	IBM DBCS Code Page 290 16-bit Japanese	MB, UDC
JA16DBCSFIXED	IBM EBCDIC 16-bit Japanese (16-bit fixed width, no single byte)	FIXED, UDC
KO16DBCS	IBM EBCDIC 16-bit Korean	MB, UDC
KO16DBCSFIXED	IBM EBCDIC 16-bit Korean (16-bit fixed-width, no single byte)	FIXED, UDC
ZHS16DBCS	IBM EBCDIC 16-bit Simplified Chinese	MB, UDC
ZHS16CGB231280 FIXED	CGB2312-80 16-bit Simplified Chinese (16-bit fixed-width, no single byte)	FIXED
ZHS16DBCSFIXED	IBM EBCDIC 16-bit Simplified Chinese (16-bit fixed-width, no single byte)	FIXED, UDC
ZHT16DBCS	IBM EBCDIC 16-bit Traditional Chinese	MB, UDC
ZHT16DBCSFIXED	IBM EBCDIC 16-bit Traditional Chinese (16-bit fixed-width, no single byte)	FIXED
KO16KSC5601	KSC5601 16-bit Korean	MB, ASCII
KO16KSCCS	KSCCS 16-bit Korean	MB, ASCII
KO16KSC5601FIXED	KSC5601 (16-bit fixed-width, no single byte)	FIXED
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
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

Table A–4 Asian Language Character Sets

Name	Description	Comments
JA16SJISFIXED	Shift-JIS 16-bit Japanese. A fixed-width subset of JA16SJIS (contains only the 2-byte characters of JA16JIS). Contains no 7- or 8-bit ASCII characters	FIXED, 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
ZHT16BIG5FIXED	BIG5 16-bit Traditional Chinese (16-bit fixed-width, no single byte)	FIXED
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
ZHT32TRISFIXED	TRIS 32-bit Fixed-width Traditional Chinese	FIXED
AL24UTFFSS	See "Universal Character Sets" on page A-17 for details	
UTF8	See "Universal Character Sets" on page A-17 for details	
UTFE	See "Universal Character Sets" on page A-17 for details	
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
WE8GCOS7	Bull EBCDIC GCOS7 8-bit West European	SB

Table A–5 European Language Character Sets

Name	Description	Comments
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.DF.L5 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
I8EBCDIC280	EBCDIC Code Page 280 / 1 8-bit Italian	SB

Table A–5 European Language Character Sets

Name	Description	Comments
WE8EBCDIC284	EBCDIC Code Page 284 8-bit Latin American / Spanish	SB
WE8EBCDIC285	EBCDIC Code Page 285 8-bit West European	SB
WE8EBCDIC1047	EBCDIC Code Page 1047 8-bit West European	SB
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
EL8EBCDIC875S	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
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
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

Table A–5 European Language Character Sets

Name	Description	Comments
I8EBCDIC1144	EBCDIC Code Page 1144 8-bit Italian	SB, EURO
F8EBCDIC1147	EBCDIC Code Page 1147 8-bit French	SB, EURO
EEC8EUROASCI	EEC Targon 35 ASCI 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
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

Table A–5 European Language Character Sets

Name	Description	Comments
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
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
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

Table A–5 European Language Character Sets

Name	Description	Comments
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
CL8KOI8R	RELCOM Internet Standard 8-bit Latin / Cyrillic	SB, ASCII
US8BS2000	Siemens 9750-62 EBCDIC 8-bit American	SB
DK8BS2000	Siemens 9750-62 EBCDIC 8-bit Danish	SB
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

Table A–5 European Language Character Sets

Name	Description	Comments
N7SIEMENS9780X	Siemens 97801/97808 7-bit Norwegian	SB
E7SIEMENS9780X	Siemens 97801/97808 7-bit Spanish	SB
S7SIEMENS9780X	Siemens 97801/97808 7-bit Swedish	SB
WE8BS2000	Siemens EBCDIC.DF.04 8-bit West European	SB
CL8BS2000	Siemens EBCDIC.EHC.LC 8-bit Cyrillic	SB
AL24UTFFSS	See "Universal Character Sets" on page A-17 for details	
UTF8	See "Universal Character Sets" on page A-17 for details	
UTFE	See "Universal Character Sets" on page A-17 for details	

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

Table A–6 Middle Eastern Character Sets

Name	Description	Comments
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 XBASIC 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
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

Table A–6 Middle Eastern Character Sets

Name	Description	Comments
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
AL24UTFFSS	See "Universal Character Sets" on page A-17 for details	
UTF8	See "Universal Character Sets" on page A-17 for details	
UTFE	See "Universal Character Sets" on page A-17 for details	

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
AL24UTFFSS	Unicode 1.1 UTF-8 Universal character set	MB, ASCII, EURO
UTF8	Unicode 2.1 UTF-8 Universal character set	MB, ASCII, EURO
UTFE	UTF-EBCDIC character set (EBCDIC-friendly UTF encoding of Unicode 2.1). UTFE works only on EBCDIC-based platforms such as IBM mainframe as compared to UTF8, which works only on ASCII-based platforms such as UNIX and Win32. The maximum length of one character in Oracle's current UTFE character set is 4 bytes (the maximum in UTF8 is 3 bytes). Both UTF8 and UTFE include exactly the same set of characters, but the character codes are different.	

Note: The Unicode 1.1 character set has been superseded by Unicode 2.1. One of the major differences between version 1.1 and 2.1 is the redefinition and addition of 11,172 Korean characters. Whenever possible, you should use the latest version of the Unicode standard. The primary scripts currently supported by Unicode 2.1 are:

Arabic	Gujarati	Latin
Armenian	Gurmukhi	Lao
Bengali	Han	Malayalam
Bopomofo	Hangul	Oriya
Cyrillic	Hebrew	Tamil
Devanagari	Hiragana	Telugu
Georgian	Kannada	Thai
Greek	Katakana	Tibetan

For details on the Unicode standard, see http://www.unicode.org or refer to the Unicode Standard, defined by the Unicode consortium.

UTF8 Support

Oracle's UTF8 character set currently supports the following characters.

- Unicode 2.1 (UCS2 and UTF16) characters U+0000 through U+007F inclusive.

  These are 1-byte characters in UTF8, that have character codes 0x00 through 0x7f inclusive. These can represent only English ASCII characters. All English ASCII characters have exactly the same character codes (0x00 through 0x7f inclusive) in US7ASCII and UTF8 character sets.

- Unicode 2.1 (UCS2 and UTF16) characters U+0080 through U+07FF inclusive

These are 2-byte characters in UTF8, that have character codes 0xc0WW through 0xdfWW inclusive where WW can be 0x80 through 0xbf inclusive.

These can represent characters of most European (including Greek and Russian), Arabic, Hebrew and some other languages.

- Unicode 2.1 (UCS2 and UTF16) characters U+0800 through U+D7FF inclusive and U+E000 through U+FFFF inclusive

  These are 3-byte characters in UTF8, that have character codes

  0xe0WWTT through 0xecWWTT inclusive

0xed80TT through 0xed9fTT inclusive

0xeeWWTT through 0xefWWTT inclusive

where WW and TT are 0x80 through 0xbf inclusive.

These can represent characters of Chinese, Japanese, Korean, Thai, Indic, Dravidian and some other languages. Also, the "euro" currency sign is included in this group of characters.

Oracle's UTF8 character set currently does not support the following characters. If you use these characters in Oracle's current UTF8 character set, the result is not guaranteed, and the behavior changes in the future releases of Oracle.

- Unicode 2.1 (UTF16) characters U+D800 through U+DFFF inclusive

  These are called surrogates in Unicode 2.1 (UTF16). These are 4-byte characters in UTF8 (when implemented in the future). Since Unicode 2.1 didn't assign any character using surrogates yet, all assigned characters in Unicode 2.1 can be represented in Oracle's current UTF8 character set. Currently, the only advantage of UTF16 (which Oracle's current UTF8 character set doesn't have) is that surrogates can represent 131,072 extra User Defined Characters on top of 6,400 User-Defined Characters that are available in Oracle's current UTF8 character set.

Therefore, unless you need more than 6,400 User-Defined Characters, Oracle's current UTF8 character set can represent all characters of Unicode 2.1.

Linguistic Definitions

Linguistic definitions define linguistic cases for particular languages. Extended linguistic definitions include some special linguistic cases for the language. Typically, using the extended definition means that characters will be sorted differently from their ASCII values. For example, *ch* and *ll* are treated as only one character in XSPANISH. Table A–8 lists the linguistic definitions supported by the Oracle server.

Table A–8 Linguistic Definitions

Basic Name	Extended Name	Special Cases
ARABIC	--	
ARABIC_MATCH	--	
ARABIC_ABJ_SORT	--	

Table A–8 Linguistic Definitions

Basic Name	Extended Name	Special Cases
ARABIC_ABJ_MATCH	--	
ASCII7	--	
BENGALI	--	
BULGARIAN	--	
CANADIAN FRENCH	--	
CATALAN	XCATALAN	æ, AE, ß
CROATIAN	XCROATIAN	D, L, N, d, l, n, ß
CZECH	XCZECH	ch, CH, Ch, ß
DANISH	XDANISH	A, ß, Å, å
DUTCH	XDUTCH	ij, IJ
EEC_EURO	--	
EEC_EUROPA3	--	
ESTONIAN	--	
FINNISH	--	
FRENCH	XFRENCH	
GERMAN	XGERMAN	ß
GERMAN_DIN	XGERMAN_DIN	ß, ä, ö, ü, Ä, Ö, Ü
GREEK	--	
HEBREW	--	
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	--	

Table A–8 Linguistic Definitions

Basic Name	Extended Name	Special Cases
LITHUANIAN	--	
MALAY	--	
NORWEGIAN	--	
POLISH	--	
PUNCTUATION	XPUNCTUATION	
ROMANIAN	--	
RUSSIAN	--	
SLOVAK	XSLOVAK	dz, DZ, Dz, ß (*caron*)
SLOVENIAN	XSLOVENIAN	ß
SPANISH	XSPANISH	ch, ll, CH, Ch, LL, Ll
SWEDISH	--	
SWISS	XSWISS	ß
THAI_DICTIONARY	--	
THAI_TELEPHONE	--	
TURKISH	XTURKISH	æ, AE, ß
UKRAINIAN	--	
UNICODE_BINARY		
VIETNAMESE	--	
WEST_EUROPEAN	XWEST_EUROPEAN	ß

Calendar Systems

By default, most territory definitions use the Gregorian calendar system. Table A–9 lists the other calendar systems supported by the Oracle server.

Table A–9 NLS Supported Calendars

Name	Default Format	Character Set Used For Default Format
Japanese Imperial	EEYY"\307\257"MM"\267\356"DD"\306\374"	JA16EUC
ROC Official	EEyy"\310\241"mm"\305\314"dd"\305\312"	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

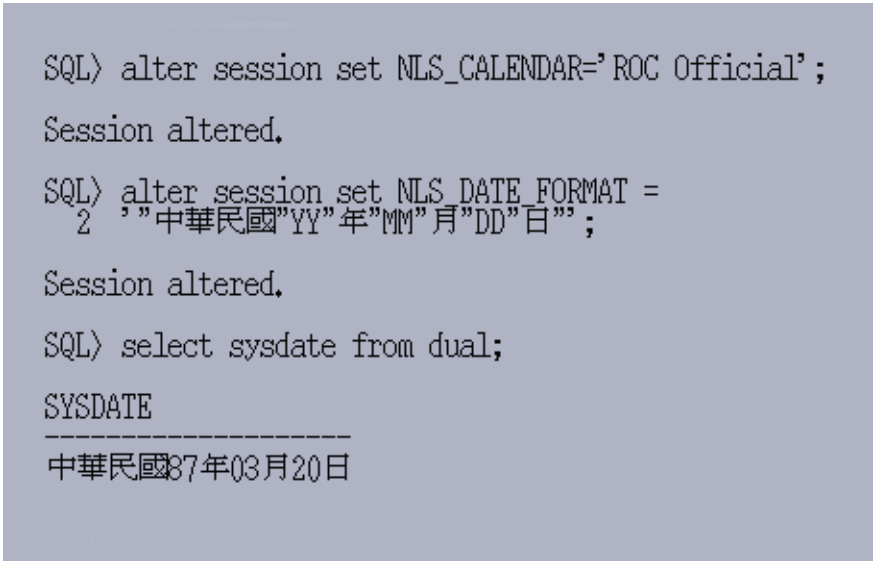
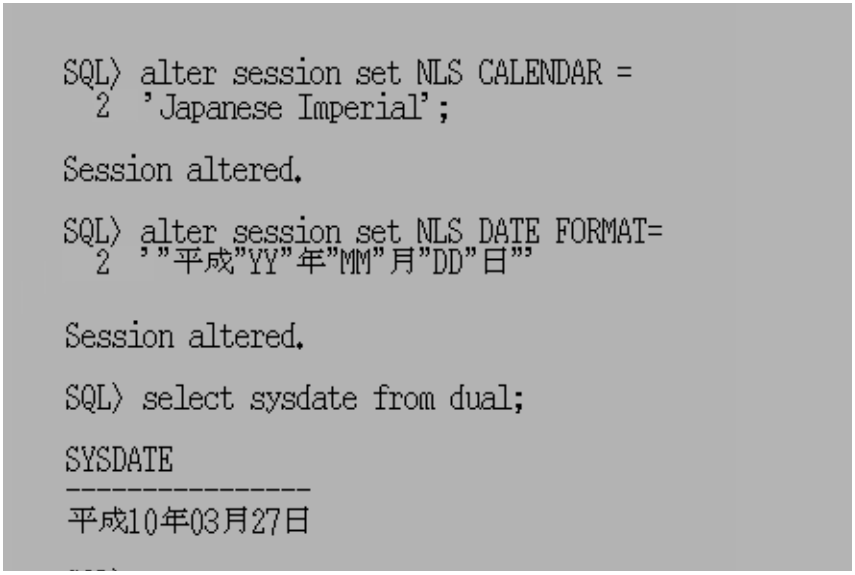


Figure A–2 shows how March 27, 1998 appears in Japanese Imperial:

Figure A–2 Japanese Imperial Example



Character Sets that Support the Euro Symbol

Table A–10 lists the character sets that support the Euro symbol.

Table A–10 Character Sets with Euro Support

Name	Description	Euro Code Value
WE8EBCDIC1140	EBCDIC Code Page 1140 8-bit West European	0x9F
WE8EBCDIC1140C	EBCDIC Code Page 1140C 8-bit West European	0x9F
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
WE8EBCDIC1145	EBCDIC Code Page 1145 8-bit West European	0x9F

Table A–10 Character Sets with Euro Support

Name	Description	Euro Code Value
WE8EBCDIC1146	EBCDIC Code Page 1146 8-bit West European	0x9F
F8EBCDIC1147	EBCDIC Code Page 1147 8-bit French	0x9F
WE8EBCDIC1148	EBCDIC Code Page 1148 8-bit West European	0x9F
WE8EBCDIC1148C	EBCDIC Code Page 1148C 8-bit West European	0x9F
WE8PC858	IBM-PC Code Page 858 8-bit West European	0xDF
EL8ISO8859P7	ISO 8859-7 Latin / Greek	0xA4
WE8ISO8859P15	ISO 8859-15 West European	0xA4
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
TR8MSWIN1254	MS Windows Code Page 1254 8-bit Turkish	0x80
IW8MSWIN1255	MS Windows Code Page 1255 8-bit Latin / Hebrew	0x80
AR8MSWIN1256	MS Windows Code Page 1256 8-bit Latin / Arabic	0x80
BLT8MSWIN1257	MS Windows Code Page 1257 Baltic	0x80
VN8MSWIN1258	MS Windows Code Page 1258 8-bit Vietnamese	0x80
TH8TISASCII	Thai Industrial 520-2533 - ASCII 8-bit	0x80
AL24UTFFSS	Unicode 1.1 UTF-8 Universal character set	U+20AC
UTF8	Unicode 2.1 UTF-8 Universal character set	U+20AC
UTFE	UTF-EBCDIC encoding of Unicode 2.1	U+20AC

Default Values for NLS Parameters

Table A–11 lists the default values for NLS parameters.

Table A–11 Default Values for NLS Parameters

Name	Default Value
NLS_CALENDAR	Gregorian
NLS_COMP	Binary
NLS_CREDIT	NLS_TERRITORY
NLS_CURRENCY	NLS_TERRITORY
NLS_DATE_FORMAT	NLS_TERRITORY
NLS_DATE_LANGUAGE	NLS_LANGUAGE
NLS_DEBIT	NLS_TERRITORY
NLS_ISO_CURRENCY	NLS_TERRITORY
NLS_LANG	American_America.US7ASCII
NLS_LANGUAGE	NLS_LANG
NLS_LIST_SEPARATOR	NLS_TERRITORY
NLS_MONETARY_ CHARACTERS	NLS_TERRITORY
NLS_CREDIT	NLS_TERRITORY
NLS_NCHAR	NLS_LANG
NLS_NUMERIC_CHARACTERS	NLS_TERRITORY
NLS_SORT	NLS_LANGUAGE
NLS_TERRITORY	NLS_LANG
NLS_DUAL_CURRENCY	NLS_TERRITORY

B

Customizing Locale Data

A set of NLS data objects is included with every Oracle distribution set, some of which is customizable.

This appendix contains:

- Customized Character Sets
- Customized Calendars
- NLS Data Installation Utility
- NLS Configuration Utility

Customized Character Sets

You can extend Oracle's character set definition files by adding user-defined characters to an existing Oracle character set.

Character set information and encoding are defined in text files. These character set definition text files contain descriptions of a character set and are specified so that a database administrator can modify or create a new character set easily. All characters are defined in terms of Unicode 2.1 code points. That is, each character is defined as a Unicode 2.1 character code value. Conversion between character sets is done by using Unicode as the intermediate form.

Once a character set definition file is created, 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 appendix allows you to convert and install character set definition text files into binary format, and merge it into an NLS data object set.

Be aware that this procedure does not ensure any of the following:

- Input of User-Defined Characters

  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

  Display of user-defined characters must still be managed by the system and/or the application. In the case of display, a new font specification may be needed. Many vendors provide support of a font editor. Once a new font is created, they must be installed onto your system and made accessible to application programs.

- Sorting of User-Defined Characters

  Sorting of user-defined characters is not supported. More specifically, customized sorting of any character set is currently not supported. Binary or linguistic sorting can be chosen, however, in the case of linguistic sorting, only the predefined Oracle linguistic sorts can be used.

Character Set Definition Files

Character set information and encoding are defined in text files (with the suffix ".nlt"). Character set definition text files (*.nlt files) contain descriptions of a character set and are specified in a user-friendly format so that a database administrator can modify or create a new character set easily. All characters are

defined in terms of Unicode 2.1 code points. That is, each character is defined as a Unicode 2.1 character code value.

Conversion between character sets is done by using Unicode as the intermediate form. The following file is a sample customized character set template character set definition file format:

Customized Character Set Definition File Format Template

```
# The following is a template of a customized character set definition file.
# You may use this template to create a user-defined character set or copy
# and modify an existing one. The convention used for naming character
# set definition (.nlt) files is in the format: lx2dddd.nlt, where
#                    dddd = 4 digit character set ID in hex
# All letters in the definition file are case-insensitive.

# Version number: specify the current loadable data version.
VERSION = <x.x.x.x.x>

# The following is the body of the definition file
DEFINE character_set

# Oracle supports a feature called 'base_char_set'. It allows you
# to extend an existing character set based on an existing Oracle supported
# standard character set. Generally, you may only need to edit the
# following fields:

# Name and ID of the character set are required for any character sets.

# Character set name must be specified in a double quoted string.
# Rules for choosing a character set name:
#          - Cannot use a character set name that is already in use. (Each
#            character set must be assigned a unique character set name).
#          - Must consist of single-byte ASCII or EBCDIC characters only
#            (single-byte compiler character set).
#          - Cannot contain multibyte characters.
#          - Maximum length of 30 characters.
#          - Must start with an alphabetic character.
#          - Composed of alphanumeric characters only (e.g. no periods,
#            dashes, underscore characters allowed)
#          - The name is case-insensitive.
# To register a unique character set name, send mail to
# nlsreg@us.oracle.com.
   name = <text_string>
```

```
# Character set ID is specified as an integer value.
# Rules for choosing a character set ID:
#         - Cannot use a character set ID that is already in use. (Each
#           character set must be assigned a unique character set ID.)
#         - Must be in the decimal range of 10000-20000
#         - Character set IDs must be registered with Oracle to receive a
#           uniquely assigned character set ID number.
# To register a unique character set ID, send mail to nlsreg@us.oracle.com.
   id   = <integer>

# The "base_char_set" feature allows users to define the base character set in
# a new character set definition file.
# The new character set will inherit all definitions from the base
# character set, therefore, the user only needs to add the customized data
# into the new character set definition file.

# The syntax of the base character set is:
#       base_char_set = <id> | <name>

#        - <id> or <name> should be a valid Oracle NLS character set id or name.
# Example is: base_char_set = "JA16EUC" or base_char_set = 830
base_char_set = <id> | <name>

# If you use base_char_set feature, remember you need to copy your base
# character set definition file (text or binary format) from $ORA_NLS33
# into the working directory specified by $ORANLS so that the new character
# set can inherit the definition from the base character set.
# Example:
#  %cp $ORA_NLS33/lx2033e.nlt $ORANLS
# or
#  %cp $ORA_NLS33/lx*33e.nlb $ORANLS


# Character data is defined as a list of  <char_value>:<unicode_value>
# pairs. <char_value> is a hex number specifying the complete character
# value in this character set (e.g. 0xa1b1), while <unicode_value> is a
# 16-bit hex number specifying its corresponding Unicode 2.1 character
# value.
# Alternatively, a range of characters can be specified with a corresponding
# range of Unicode values. Each successive character in the
# <start_char>-<end_char> range will be assigned to each successive
# character in the <start_unicode>-<end_unicode> range. There must be
# an equal number of characters in each range.
# User-defined characters must be assigned to characters in Unicode's
# private use area, and in particular the range 0xe000 to 0xf4ff. The
```

```
# remaining 1024 characters in the private use area are reserved for Oracle
# private use.
# If you already defined "base_char_set", you only need to add the
# customized character set mappings.
   character_data = {
<char_value>:<unicode_value>,
<start_char>-<end_char>:<start_unicode>-<end_unicode>,
...
   }

# A character classification list is used to specify the type of characters.
# Valid values:
# UPPER LOWER DIGIT SPACE PUNCTUATION CONTROL
#            HEX_DIGIT LETTER PRINTABLE
# You only need to add customized characters' classification if you defined
# base_char_set.
classification = {
<char_value> = { UPPER, LOWER, DIGIT,
       SPACE, PUNCTUATION, CONTROL,
       HEX_DIGIT, LETTER, PRINTABLE },
...
   }

# Lower-to-Upper case character relationships are defined as pairs, where
# the first specifies the value of a character in this character set and the
# second specifies its uppercase value in this character set.  You may add
# the customized case mapping only if needed.
   uppercase = {
<char_value>:<upper_char_value>,
<start_char>-<end_char>:<start_upper>-<end_upper>,
...
   }

# Upper-to-Lower case character relationships are defined as pairs, where
# the first specifies the value of a character in this character set and the
# second specifies its lowercase value in this character set. You may add
# the customized case mapping only if needed.
   lowercase = {
<char_value>:<lower_char_value>,
<start_char>-<end_char>:<start_lower>-<end_lower>,
...
   }

# There are a lot of other fields in an Oracle character set definition file.
# Presumably, you will only need the above fields, at most.
```

```
ENDDEFINE character_set
```

Example of Character Set Customization

This section uses an example to introduce the steps required to create a new character set with an example. For this example, we will create a new character set based on Oracle's JA16EUC character set and add a few user defined characters.

Step 1. Register a New Character Set Name and ID

In order to maintain unique character set names and IDs, you must register the character name with Oracle to receive a uniquely assigned character set ID. Requests for character set name and ID registration can be sent to:

`nlsreg@us.oracle.com`

> **Note:** If the character set name and ID are not unique, you could experience incompatibilities between character sets and potential loss of data.

Observe the following restrictions on character set names:

- you cannot use a character set name that is already in use. (Each character set must be assigned a unique character set name)
- the name must consist of single-byte ASCII or EBCDIC characters only (single-byte compiler character set)
- there is a maximum length of 30 characters
- the name must start with an alphabetic character
- the name must be composed of alphanumeric characters only (e.g., no periods, dashes, underscore characters allowed)
- the name is case-insensitive

Rules for choosing a character set ID:

- the ID cannot use a character set ID that is already in use (each character set must be assigned a unique character set ID)
- the ID must be in the decimal range of 10000-20000 (hexadecimal range of 0x2710-0x3a98)

If a character set is derived from an existing Oracle character set, we recommend using the following character set naming convention:

```
<Oracle_character_set_name><organization_name>EXT<version>
```

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

For this example and all further steps, we will use the character set ID 10000 (hex value 0x2710).

Step 2. Create an NLS Text Boot File

The NLS binary boot files indicate which NLS data objects will be loaded into the database. Therefore, the binary boot file must be updated whenever a new character set is created. To update the binary boot file, you must create an entry for your new character set in a text boot file lx0boot.nlt first.

NLS Boot File Format

```
# The following is a template for an Oracle NLS boot file.

# Version number specifies the current loadable data version.
VERSION=<x.x.x.x.x>

# List the character set names and IDs that will be merged into the existing
# system boot file using the $ORACLE_HOME/bin/lxinst utility.
#
CHARACTER_SET
<name> <id>
```

```
<name> <id>
...
```

Example:

Create a text boot file (lx0boot.nlt) in the working directory.

```
% vi /tmp/lx0boot.nlt
```

To add JA16EUCSUNWEXT1, set:

```
VERSION=2.1.0.0.0

CHARACTER_SET
"JA16EUCSUNwEXT1"  10000
```

where the version number is based on the Oracle release. Refer to the version number listed in the existing lx2*.nlt files for the latest version number.

Note that it is possible to list multiple user defined character sets in a single lx0boot.nlt file. For example:

```
VERSION=2.1.0.0.0

CHARACTER_SET
"JA16EUCSUNwEXT1"  10000
"ZH16EUCSUNwEXT1"  10001
```

Step 3. Create a Character Set Definition File (lx2*dddd*.nlt)

The convention used for naming character set definition (.nlt) files is in the format: lx2*dddd*.nlt, where *dddd* = 4 digit Character Set ID in hex.

A few things to note when editing a character set definition file:

- You can only extend (add characters to) an existing Oracle character set.
- You should not remap existing characters.
- All character mappings must be unique.
- One-to-many character mapping is not allowed.
- Many-to-one character mapping is not allowed.
- New characters should be mapped into the Unicode private use range: e000-f4ff. (Note that the actual Unicode 2.1 private use range is e000-f8ff, however, Oracle reserves f500-f8ff for its own private use.)
- No line can be longer than 80 characters in the character set definition file.

There is a feature, 'BASE_CHAR_SET', that can make customized character set support easier. Since you are extending an existing Oracle character set, you can use the 'BASE_CHAR_SET' feature which causes the new character set to inherit all definitions from the base character set and the user only need add user-specific customized character set data.

Example:

Assume you are extending the JA16EUC character set and have added some new customized character set data to it.

Based on the character set ID of 10000 you specified in Step 1, name the new character set definition file lx*22710*.nlt (based on the character set id hex value of 0x2710).

This example uses **/tmp** as the working directory. Edit the new character definition file with an editor.

```
% vi /tmp/lx22710.nlt
VERSION = 2.1.0.0.0

DEFINE character_set
  name = "JA16EUCSUNWEXT1"
  id = 10000
  base_char_set = 830
  character_data = {
     0x9a41 : 0xe001,
     0x9a42 : 0xe002,
  }
  classification = {
     0x9a41 = { LETTER, LOWER },
     0x9a42 = { LETTER, UPPER },
  }
  uppercase = {
     0x9a41 : 0x9a42,
  }
  lowercase = {
     0x9a42 : 0x9a41,
  }
ENDDEFINE character_set
```

Refer to "Customized Character Set Definition File Format Template" on page B-3 for more information about the format of the character set definition files.
Minimally, you will need to set the character set name, character set ID and, base character set, add customized character data and classification fields.

Step 4. 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 prior to generating and installing .nlb files.

```
% cd $ORA_NLS33
% cp lx0boot.nlb lx0boot.nlb.orig
% cp lx1boot.nlb lx1boot.nlb.orig
```

Step 5. 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 you must make sure to regenerate them on each platform and also install these files on both the server and clients.

You use the lxinst utility to create both the binary character definition files (lx2dddd.nlb) and update the NLS boot file (lx*boot.nlb).

Example:

The lxinst utility will make use of the existing system boot file. Therefore, copy the existing binary system boot file into the directory specified by SYSDIR. For this example, specify SYSDIR to the working directory (/tmp).

```
% 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 Step 2 & 3 should reside in the directory specified by ORANLS, for this example, specify it to be /tmp. Also, since we define JA16EUC (Id 830 in hex value 033e) as "BASE_CHAR_SET", the base definition file, text-format (lx2033e.nlt) or binary format (lx*033e.nlb), should be in the directory ORANLS too, so that the new character set can inherit all definitions from it.

```
% cp lx2033e.nlt /tmp
```

or

```
% cp lx*033e.nlb /tmp
```

Use the lxinst utility to generate a binary character set definition file (lx22710.nlb) in the directory specified by 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.

```
% $ORACLE_HOME/bin/lxinst oranls=/tmp sysdir=/tmp destdir=/tmp
```

Then, 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 is lx5*.nlb or lx6*.nlb or both, install them too:

```
% cp /tmp/lx22710.nlb $ORA_NLS33
% cp /tmp/lx52710.nlb $ORA_NLS33
% cp /tmp/lx62710.nlb $ORA_NLS33
```

Step 6. Repeat for Each Platform

You must repeat Step 5 on each hardware platform since 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 7. Create the Database Using New Character Set

After installing the .nlb files, you must shutdown 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.

Customized 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

Syntax

The Calendar Utility is invoked directly from the command line:

```
LXEGEN
```

There are no parameters.

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. Please see the 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 would be lxecal.nlb were you to define deviation days for the Arabic Hijrah calendar. 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.

Utilities

The Oracle server includes the following three utilities to assist you in maintaining NLS data:

NLS Data Installation Utility (lxinst)	Generate binary-format data objects from their text-format versions. Use this when you receive NLS data updates or if you create your own data objects.
NLS Calendar Utility (lxegen)	Generate a binary file with the appropriate format for the calendar data.
NLS Configuration Utility (lxbcnf)	Create and edit user boot files.

NLS Data Installation Utility

Overview

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 Oracle8*i*, 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 described here will allow 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. For information about how this file is generated, see "NLS Configuration Utility" on page B-16.

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=pathname	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=pathname	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=pathname	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 "Data Object File Names" on page B-16 for naming convention information.

  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.

Note: When you want to create a new character set, you must register with Oracle Corporation by sending email to nlsreg@us.oracle.com, which will ensure that your character set has a unique name and ID.

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:

lxtdddd

where:

t	1 digit object type (hex)
dddd	4 digit object ID (hex)

The installation boot file name is *lx*0BOOT; the system boot file name is *lx*1BOOT; user boot files are named *lx*2BOOT. The file extension for text format files is .nlt, for binary files, .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

NLS Configuration Utility

At installation, all available NLS objects are stored and referenced in the system boot file. This file is used to load the available NLS data.

The NLS Configuration Utility allows you to configure your boot files such that only the NLS objects that you require will be loaded. It does this by creating a user boot file, which contains a subset of the system boot file. Data loading by the kernel will then be performed according to the contents of this user boot file.

The NLS Configuration Utility allows you to configure a user boot file, either by selecting NLS objects from the installed system boot file which will then be included in a new user boot file, or by reading entries from an existing user boot file and possibly removing one or more of them and saving the remaining entries into a new user boot file. Note that you will not be allowed to actually "edit" an existing boot file as it may be in use by either the RDBMS or some other Oracle tool (that is, saving of boot file entries is never done to an existing one).

You may also use the NLS Data Installation Utility to check the integrity of an existing user boot file. This is necessary since the contents of existing NLS objects may change over time, and the installation of a new system boot file may cause user boot files to become out of date. Thus, a comparison function will notify you when it finds that the file is out of date and will allow you to create a new user boot file.

Syntax

The NLS Configuration Utility is invoked from the command line with the following syntax:

```
LXBCNF [ORANLS=pathname] [userbootdir=pathname] [DESTDIR=pathname]
[HELP=[yes | no]]
```

where:

ORANLS=pathname	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, the NLS Installation Utility uses the value in the environment variable ORA_NLS (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=pathname	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 or move it to the appropriate directory.
DESTDIR=pathname	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.

Menus

When the NLS Configuration Utility is started you are presented with the following top-level menu:

- File Menu
- Edit Menu
- Action Menu
- Windows Menu
- Help

File Menu

The file menu contains choices pertaining to file operations. Options are:

Table B–1 File Menu Options

Menu Item	Options	Description
System Boot File	Open	This will open the current system boot file. Note that the Open menu item will be "greyed out" as soon as a system Boot File has been successfully read. Also note that you cannot perform any other functions until you have opened a system boot file.
User Boot File	New	Open a new user boot file.
	Read	Read the contents of an existing user boot file.
	Save	Save changes to the new user boot file.
	Revert	Undo the changes to the currently open user boot file made since the last "Save".
Choose Printer		Not implemented in this release.
Page Setup		Not implemented in this release.
Print		Not implemented in this release.
Quit		Exit from the file.

Note: As long as the system boot file has not been opened and read, all these menu items will remain *greyed out*. That is, you cannot build a user boot file as long as there is no system boot file information available.

As soon as you select New to create a new user boot file, the following NLS objects will be created in the new file by default:

If you choose to read the contents of an existing user boot file, the entries read will be checked against the entries of the system boot file. If an entry is found which does not exist in the system boot file, you will receive a warning, and the entry will not be included.

Edit Menu

The Edit Menu contains choices for editing information that you enter in any of the dialogs or windows of the NLS Configuration Utility.

Action Menu

The Action Menu contains choices for performing operations on the user boot file. Note that this menu is available only in the character mode NLS Configuration Utility.

Copy Item	Copies the selected item from the system boot file to the user boot file.
Delete Item	Deletes the selected item from the user boot file.

Windows Menu

The Windows Menu allows you to either activate certain windows or set the focus to an already open window (the latter is meant for character-mode platforms). Whenever a new window is opened, its name will be added to the Windows Menu automatically.

NLS Defaults	Not implemented in this release.

Help Menu

This menu provides functions which allow you to retrieve various levels of help about the NLS Configuration Utility.

About	Shows version information for the NLS Configuration Utility.
Help System	Not implemented in this release.

C

Obsolete Locale Data

Oracle has renamed many character sets over time. This appendix lists them in:

- Obsolete NLS Data

Obsolete NLS Data

Prior to 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 C–1 lists the affected character sets. If you reference any of these character sets in your code, please replace them with their new name:

Table C–1 New Names for Obsolete NLS Data Character Sets

Old Name	New Name
AR8MSAWIN	AR8MSWIN1256
JVMS	JA16VMS
JEUC	JA16EUC
SJIS	JA16SJIS
JDBCS	JA16DBCS
KSC5601	KO16KSC5601
KDBCS	KO16DBCS
CGB2312-80	ZHS16CGB231280
CNS 11643-86	ZHT32EUC
ZHT32CNS1164386	ZHT32EUC

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.

D

Glossary

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.

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, A, A, etc. All these different forms are different glyphs, but representing the same character. A character, a character code and a glyph are related as follows.

character --(encoding)--> character code --(font)--> glyph

When we have the first character of the English upper-case Alphabet in computer memory, we actually have 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 upper-case 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 "character" also.)

Character Set

A character set is a set of characters for a specific language (or languages). There can be many different character sets just for one language.

Sometimes, a character set doesn't imply any specific character encoding scheme.

In this manual, a character set generally implies a specific character encoding scheme, which is how a number (or a character code) is assigned to each character of the character set. Therefore, the meaning of the term character set is generally same as encoded character set in this manual.

Character String

A character string is a serial string of characters.

A character string can also consist of no character. In this case, the character string doesn't include any character. This character string is called "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).

Character Set Conversion

Conversion from one encoded character set to another.

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 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.

DBCS

DBCS stands for Double-Byte (Coded) Character Set. However, this term should be used carefully. (Use the term multibyte (coded) character set when appropriate.) See "double-byte" also.

Double byte

Double-byte (or doublebyte or double byte) means two bytes. However, this term should be used carefully. (Use the term multibyte when appropriate.) For many characters of many languages, double-byte is not enough (this is especially true for UTF8 encoding of Unicode).

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. Same as Character Encoding Scheme.

Encoding Scheme

See "Character Encoding Schemes".

EUC

Extended UNIX Codes. A common encoding method used on Asian UNIX systems. It combines up to four different encoded character sets in a single data stream.

Euro

The new unit of currency used by participating member states of the European Union.

Font

An ordered collection of character glyphs which provides a graphical representation of characters within a character set.

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, A, A, etc.

All these different forms are different glyphs, but representing the same character. (See "character" also.)

Ideograph

A symbol representing an idea. Chinese is an example of an ideographic 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 Standards Organization.

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: UCS2 is a 2-byte fixed-width format and UCS4 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.

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 also "ISO 8859".

Linguistic Index

An index built on a linguistic collation order.

Linguistic Sorting

Sorting of strings based on requirements from a locale instead of based on the binary representation of the strings.

Local Currency

The currency symbol used in a country or region. For example, "$" represents the United States Dollar.

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.

Localization

The process of providing language- 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 Support

Support for only one language.

Multibyte

Multi-byte (or multibyte or multi byte) means two or more bytes.

When we assign character codes to all characters for 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 UTF8 encoding of Unicode. In UTF8, there are a lot of 2-byte and 3-byte characters.

Another example is Traditional Chinese language used in Taiwan. It has more than 80,000 different characters. We are using 4 bytes for some of those characters under some character encoding schemes used in Taiwan.

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 we are using, we 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" also.

Multibyte Character String

A multibyte character string is a character string which consists of one of the below.

- No character

  (The character string is called "null string" in this case.)
- One or more single-byte character(s)
- A mixture of one or more single-byte character(s) and one or more multibyte character(s)
- One or more multibyte character(s)

Theoretically, we can exclude single-byte character strings (character strings including only single-byte characters) from the list above. However, it's probably more convenient for software to handle single-byte character strings as one type of multibyte character strings.

NCHAR Character Set

An alternate character set from the database character set that can be specified for NCHAR, NVARCHAR2, and NCLOB columns. NCHAR character sets, unlike the database character set, can support fixed-width multibyte character sets. Care must be taken when selecting an NCHAR character set, since its character repertoire must be included in the database character set as well.

Net8

Net8 enables two or more computers that run the Oracle server to exchange data through a third-party network. It is independent of the communications protocol.

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.

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 (i.e., NLSDATA) is read by the NLSRTL library during run-time.

Replacement Character

A character used during character conversion when the desired 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*Net

Now called Net8. Net8 enables two or more computers that run the Oracle server to exchange data through a third-party network. It is independent of the communications protocol.

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.

Server Character Set

The character set used by the database server.

Single-byte

Single-byte (or singlebyte 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 UCS2, and three bytes in UTF8. See "multibyte character" also.

Single-byte Character String

A single-byte character string is a character string which consists of one of the below.

- No character

  (The character string is called "null string" in this case.)

- One or more single-byte character(s).

UCS-2

UCS stands for "Universal Multiple-Octet Coded Character Set". It is a 1993 ISO and IEC standard character set. See "UCS2".

UCS2

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.

UCS4

Fixed-width 32-bit Unicode. Each character occupies 32 bits of storage. The UCS2 characters are the first 65,536 code points in this standard, so it can be viewed as a 32-bit extension of UCS2. 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 UCS2 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 is interchangeable with code element, code position, and code value.

Unicode Mapping Between UCS and UTF Formats

The following shows how different Unicode-related character sets relate to one another in terms of character code value ranges:

UCS2	UTF8	Description
0x0000 - 0x007F	0x00 - 0x7F	Single bytes
0x0080 - 0x07FF	0xC0 - 0xDF	2-byte sequence leaders (5+6 bits)
0x0800 - 0xFFFF	0xE0 - 0xEF	3-byte sequence leaders (4+6+6 bits)
	0x80 - 0xBF	Follower bytes (6 bits each)

UCS4	UTF8	Description
0x00000000 - 0x0000007F	0x00 - 0x7F	Single bytes
0x00000080 - 0x000007FF	0xC0 - 0xDF	2-byte sequence leaders (5+6 bits)

0x00000800 - 0x0000FFFF	0xE0 - 0xEF	3-byte sequence leaders (4+6+6 bits)
0x00001000 - 0x001FFFFF	0xF0 - 0xF7	4-byte sequence leaders (3+6+6+6 bits)
0x00200000 - 0x03FFFFFF	0xF8 - 0xFB	5-byte sequence leaders (2+6+6+6+6 bits)
0x04000000 - 0x7FFFFFFF	0xFC - 0xFD	6-byte sequence leaders (1+6+6+6+6+6 bits)
	0x80 - 0xBF	Follower bytes (6 bits each)
	0xFE - 0xFF	Reserved or unused

UCS4	UTF16	Description
0x00000000 - 0x0000FFFF	0x0000 - 0xFFFF	Same as UCS2
0x00010000 - 0x0010FFFF	0xD800 - 0xDBFF	High surrogate ((x-0x10000)>>10)&0x3FF
	0xDC00 - 0xDFFF	Low surrogate (x-0x10000)&0x3FF
0x00110000 - 0x7FFFFFFF		Not mapped to UTF16

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.

UTF-8

A variable-width encoding of UCS2 which 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 (for the Unicode 2.1 standard). 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 UCS2 that allows for pairs of UCS2 code points to represent extended characters from the UCS4 set. UCS2 has ranges of code points allocated for high (leading) and low (trailing) surrogates that support UTF16 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

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W