Oracle9*i* Database

Globalization Support Guide

Release 2 (9.2)

March 2002

Part No. A96529-01

Oracle9*i* Database Globalization Support Guide, Release 2 (9.2)

Part No. A96529-01

Primary Author: Cathy Baird

Contributors: Dan Chiba, Winson Chu, Jessica Fan, Claire Ho, Simon Law, Geoff Lee, Peter Linsley, Keni Matsuda, Tamzin Oscroft, Shige Takeda, Linus Tanaka, Makoto Tozawa, Barry Trute, Mayumi Tsujimoto, Ying Wu, Michael Yau, Tim Yu, Chao Wang, Simon Wong, Weiran Zhang, Lei Zheng, Yan Zhu

Graphic Designer: Valarie Moore

Contents

Send Us Your Comments

Oracle9*i* Database Globalization Support Guide, Release 2 (9.2)

Part No. A96529-01

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

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

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

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227   Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

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

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

Preface

This manual describes Oracle’s globalization support and how to use its capabilities.

This preface contains these topics:

Audience

Oracle9*i* Database Globalization Support Guide is intended for database administrators, system administrators, and database application developers who perform the following tasks:

- Set up a globalization support environment
- Choose, analyze, or migrate character sets
- Sort data linguistically
- Customize locale data
- Write programs in a global environment
- Use Unicode

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

Organization

This document contains:

Chapter 1, "Overview of Globalization Support"

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

Chapter 2, "Choosing a Character Set"

This chapter describes how to choose a character set.

Chapter 3, "Setting Up a Globalization Support Environment"

This chapter contains sample scenarios for enabling globalization capabilities.

Chapter 4, "Linguistic Sorting"

This chapter describes linguistic sorting.

Chapter 5, "Supporting Multilingual Databases with Unicode"

This chapter describes Unicode considerations for databases.

Chapter 6, "Programming with Unicode"

This chapter describes how to program in a Unicode environment.

Chapter 7, "SQL and PL/SQL Programming in a Global Environment"

This chapter describes globalization considerations for SQL programming.

Chapter 8, "OCI Programming in a Global Environment"

This chapter describes globalization considerations for OCI programming.

Chapter 9, "Java Programming in a Global Environment"

This chapter describes globalization considerations for Java.

Chapter 10, "Character Set Migration"

This chapter describes character set conversion issues and character set migration.

Chapter 11, "Character Set Scanner"

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

Chapter 12, "Customizing Locale Data"

This chapter explains how to use the Oracle Locale Builder utility to customize locales. It also contains information about time zone files and customizing calendar data.

Appendix A, "Locale Data"

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

Appendix B, "Unicode Character Code Assignments"

This chapter lists Unicode code point values.

Glossary

The glossary contains definitions of globalization support terms.

Related Documentation

For more information, see these Oracle resources:

- *Oracle9i SQL Reference*
- *Oracle9i Application Developer's Guide - Fundamentals*

Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them yourself.

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

`http://oraclestore.oracle.com/`

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

`http://www.oraclebookshop.com/`

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

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

`http://otn.oracle.com/admin/account/membership.html`

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

`http://otn.oracle.com/docs/index.htm`

To access the database documentation search engine directly, please visit

`http://tahiti.oracle.com`

Conventions

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

- Conventions in Text
- Conventions in Code Examples
- Conventions for Windows Operating Systems

Conventions in Text

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

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true.` Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

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

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

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

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	DECIMAL (*digits* [, *precision*])
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	{ENABLE | DISABLE}
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]
...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	CREATE TABLE ... AS *subquery*;
	■ That you can repeat a portion of the code	SELECT *col1*, *col2*, ... , *coln* FROM employees;
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	SQL> SELECT NAME FROM V$DATAFILE; NAME ------------------------------------ /fsl/dbs/tbs_01.dbf /fs1/dbs/tbs_02.dbf . . . /fsl/dbs/tbs_09.dbf 9 rows selected.

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	acctbal NUMBER(11,2);
		acct CONSTANT NUMBER(4) := 3;

Convention	Meaning	Example

Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`*
		`DB_NAME = `*`database_name`*

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Conventions for Windows Operating Systems

The following table describes conventions for Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose Start >	How to start a program.	To start the Database Configuration Assistant, choose Start > Programs > Oracle - *HOME_NAME* > Configuration and Migration Tools > Database Configuration Assistant.
File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	`c:\winnt"\"system32` is the same as `C:\WINNT\SYSTEM32`

Convention	Meaning	Example
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`

Special characters	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (’) do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\>exp scott/tiger TABLES=emp QUERY=\"WHERE job=’SALESMAN’ and sal<1600\"`
		 `C:\>imp SYSTEM/`*`password`*` FROMUSER=scott TABLES=(emp, dept)`

`HOME_NAME`	Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`C:\> net start Oracle`*`HOME_NAME`*`TNSListener`

Convention	Meaning	Example
ORACLE_HOME and *ORACLE_ BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory that by default used one of the following names: ▪ `C:\orant` for Windows NT ▪ `C:\orawin98` for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is `C:\oracle`. If you instal the latest Oracle release on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is `C:\oracle\ora`*nn*, where *nn* is the latest release number. The Oracle home directory is located directly under *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle9i Database Getting Starting for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*`\`*ORACLE_ HOME*`\rdbms\admin` directory.

Documentation Accessibility

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

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

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

What’s New in Globalization Support?

This section describes new features of globalization support in Oracle9*i* release 2 (9.2) and provides pointers to additional information.

The following section describes the new features in Oracle globalization support:

- Oracle9i Release 2 (9.2) New Features in Globalization Support

Oracle9*i* Release 2 (9.2) New Features in Globalization Support

- **Unicode 3.1 support**

  Oracle9*i* release 2 (9.2) supports Unicode 3.1.

  > **See Also:** Chapter 5, "Supporting Multilingual Databases with Unicode"

- **ALTER TABLE MODIFY statement**

  The `ALTER TABLE MODIFY` statement can be used to change column definitions from the `CHAR` datatypes to the `NCHAR` datatypes. It also converts the data in the columns from the `CHAR` datatypes to the `NCHAR` datatypes.

  > **See Also:** "Using the ALTER TABLE MODIFY Statement to Change CHAR Columns to NCHAR Columns" on page 10-13

- **Oracle Locale Builder enhancements**

  You can view and print code charts with Oracle Locale Builder.

  Characters with diacritics can now be viewed in collation trees.

  > **See Also:**
  > - "Displaying a Code Chart with the Oracle Locale Builder" on page 12-18
  > - "Creating a New Linguistic Sort with the Oracle Locale Builder" on page 12-35

- **Character Set Scanner enhancements**

  Two new parameters have been added to the Character Set Scanner: `EXCLUDE` and `PRESERVE`.

  The `TABLE` parameter has been extended to support multiple tables.

  Convertible and exceptional data dictionary data are documented in the new "Data Dictionary Individual Exceptions" of the Individual Exception Report.

  > **See Also:**
  > - "Character Set Scanner Parameters" on page 11-9
  > - "Individual Exception Report" on page 11-27

- **Change in euro support**

  The members of the European Monetary Union (EMU) use the euro as their currency as of January 1, 2002. Setting NLS_TERRITORY to correspond to a country in the EMU (Austria, Belgium, Finland, France, Germany, Greece, Ireland, Italy, Luxembourg, the Netherlands, Portugal, and Spain) results in the default values for NLS_CURRENCY and NLS_DUAL_CURRENCY being set to EURO. Beginning with Oracle9*i* release 2 (9.2), the value of NLS_ISO_CURRENCY results in the ISO currency symbol being set to EUR for EMU member countries.

  See Also: "Oracle Support for the Euro" on page 3-37

- **OCIEnvNlsCreate() function**

  Use the OCIEnvNlsCreate function to specify client-side database and national character sets when the OCI environment is created.This function allows users to set character set information dynamically in applications, independent of the NLS_LANG and NLS_CHAR initialization parameter settings. In addition, one application can initialize several environment handles for different client environments in the same server environment.

  See Also: "OCIEnvNlsCreate()" on page 8-2

- **OCINlsCharSetConvert() function**

  This function converts a string from one character set to another.

  See Also: "OCINlsCharSetConvert()" on page 8-50

- **OCINlsCharSetNameTold() function**

  This function returns the Oracle character set ID for the specified Oracle character set name.

  See Also: "OCINlsCharSetNameTold()" on page 8-10

- **OCINlsCharSetIdToName() function**

  This function returns the Oracle character set name from the specified character set ID.

  See Also: "OCINlsCharSetIdToName()" on page 8-11

- **OCINlsNumericInfoGet() function**

  This function generates numeric language information specified by item from the OCI environment handle into an output number variable.

  > **See Also:** "OCINlsNumericInfoGet()" on page 8-12

- **OCINlsNameMap() function**

  This function maps Oracle character set names, language names, and territory names to and from IANA and ISO names.

  > **See Also:** "OCINlsNameMap()" on page 8-14

- **DBMS_LOB.LOADBLOBFROM FILE and DBMS_LOB.LOADCLOBFROM FILE**

  These APIs allow the user to specify the character set ID of `BFILE` data by using a new parameter. The APIs convert the data from the specified `BFILE` character set into the database character set for `CLOB`s or the national character set for `NCLOB`s.

  > **See Also:** "Loading External BFILE Data into LOBs"‘ on page 7-15

- **Generic base letter search**

  You can perform a search that ignores case and diacritics.

  > **See Also:** "Performing a Generic Base Letter Search" on page 4-16

- **Change in Object Types support for NCHAR datatypes and character semantics**

  Object Types now support `NCHAR` datatypes and character semantics.

  > **See Also:**
  >
  > - "Length Semantics" on page 2-2
  > - "Summary of Supported Datatypes" on page 2-19
  > - *Oracle9i Application Developer's Guide - Object-Relational Features*

1 Overview of Globalization Support

This chapter provides an overview of Oracle globalization support. It includes the following topics:

- Globalization Support Architecture
- Globalization Support Features

Globalization Support Architecture

Oracle's globalization support enables you to store, process, and retrieve data in native languages. It ensures that database utilities, error messages, sort order, and date, time, monetary, numeric, and calendar conventions automatically adapt to any native language and locale.

In the past, Oracle's globalization support capabilities were referred to as National Language Support (NLS) features. National Language Support is a subset of globalization support. National Language Support is the ability to choose a national language and store data in a specific character set. Globalization support enables you to develop multilingual applications and software products that can be accessed and run from anywhere in the world simultaneously. An application can render content of the user interface and process data in the native users' languages and locale preferences.

Locale Data on Demand

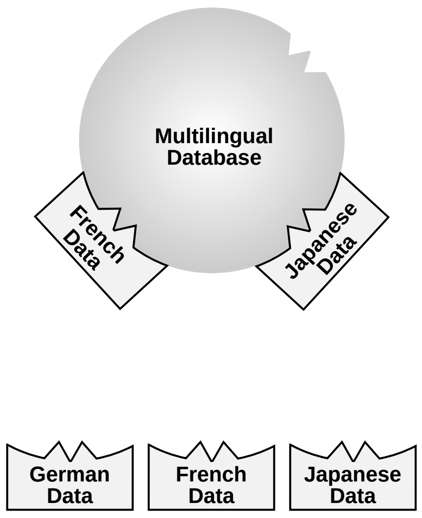
Oracle's globalization support is implemented with the Oracle NLS Runtime Library (NLSRTL). The NLS Runtime Library provides a comprehensive suite of language-independent functions that allow proper text and character processing and language convention manipulations. Behavior of these functions for a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

The locale-specific data is structured as independent sets of data for each locale that Oracle supports. The data for a particular locale can be loaded independent of other locale data. The advantages of this design are as follows:

- You can manage the memory consumption of Oracle9*i* by choosing the set of locales that you need.
- You can add and customize locale data for a specific locale without affecting other locales.

Figure 1–1 shows that locale-specific data is loaded at runtime. In this example, French data and Japanese data are loaded into the multilingual database, but German data is not.

Figure 1–1 Loading Locale-Specific Data to the Database



The locale-specific data is stored in a directory specified by the `ORA_NLS*` environment variable. There is a different `ORA_NLS` data directory for different releases of the Oracle database server. For Oracle9*i*, the `ORA_NLS33` directory is used. Table 1–1 shows the environment variable that specifies the location of locale-specific data for different releases of the Oracle database server.

Table 1–1 Environment Variable that Specifies Location of Locale-Specific Data by Release

Release	Environment Variable
7.2	`ORA_NLS`
7.3	`ORA_NLS32`
8.0, 8.1, 9.0.1, 9.2	`ORA_NLS33`

When the `ORA_NLS*` environment variable is not defined, then the default value relative to the Oracle home directory is used to locate the locale-specific data. The default location of locale data is `$ORACLE_HOME/ocommon/nls/admin/data` in all releases. In most cases, the default value is sufficient. The `ORA_NLS*` variable should be defined only when the system has multiple Oracle homes that share a single copy of NLS datafiles.

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 are available for the database. Also, new locale data can be added and some locale data components can be customized.

See Also: Chapter 12, "Customizing Locale Data" for more information about customizing locale data

Architecture to Support Multilingual Applications

The Oracle9*i* database is implemented to enable multitier applications and client/server applications to support languages for which the database is configured.

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

The database has a set of session-independent NLS parameters that are specified when the database is created. Two of the parameters specify the database character set and the national character set, an alternate Unicode character set that can be specified for `NCHAR`, `NVARCHAR2`, and `NCLOB` data. The parameters specify the character set that is used to store text data in the database. Other parameters, like language and territory, are used to evaluate check constraints.

If the client session and the database server specify different character sets, then the Oracle9*i* database converts character set strings automatically.

From a globalization support perspective, all applications are considered to be clients, even if they run on the same physical machine as the Oracle instance. For example, when SQL*Plus is started by the UNIX user who owns the Oracle software from the Oracle home in which the RDBMS software is installed, and SQL*Plus connects to the database through an adapter by specifying the `ORACLE_SID` parameter, SQL*Plus is considered a client. Its behavior is ruled by client-side NLS parameters.

Another example of an application being considered a client occurs when the middle tier is an application server. The different sessions spawned by the application server are considered to be separate client sessions.

When a client application is started, it initializes the client NLS environment from environment settings. All NLS operations performed locally are executed using these settings. Examples of local NLS operations are:

- Display formatting in Oracle Developer applications
- User OCI code that executes NLS OCI functions with OCI environment handles

When the application connects to a database, a session is created on the server. The new session initializes its NLS environment from NLS instance parameters specified in the initialization parameter file. These settings can be subsequently changed by an `ALTER SESSION` statement. The statement changes only the session NLS environment. It does not change the local client NLS environment. The session NLS settings are used to process SQL and PL/SQL statements that are executed on the server. For example, use an `ALTER SESSION` statement to set the `NLS_LANGUAGE` initialization parameter to Italian:

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

Enter a `SELECT` statement:

```
SQL> SELECT last_name, hire_date, ROUND(salary/8,2) salary FROM employees;
```

You should see results similar to the following:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
Sciarra                   30-SET-97      962.5
Urman                     07-MAR-98        975
Popp                      07-DIC-99      862.5
```

Note that the month name abbreviations are in Italian.

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

See Also:

- Chapter 8, "OCI Programming in a Global Environment"
- Chapter 3, "Setting Up a Globalization Support Environment"

Using Unicode in a Multilingual Database

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

When Unicode is used in an Oracle9*i* database, it has the following advantages:

- Simplifies character set conversion and linguistic sort functions
- Improves performance compared with native multibyte character sets
- Supports the Unicode datatype based on the Unicode standard

See Also:

- Chapter 5, "Supporting Multilingual Databases with Unicode"
- Chapter 6, "Programming with Unicode"
- "Enabling Multilingual Support with Unicode Datatypes" on page 5-8

Globalization Support Features

Oracle's standard features include:

- Language Support
- Territory Support
- Date and Time Formats
- Monetary and Numeric Formats
- Calendars Feature
- Linguistic Sorting
- Character Set Support
- Character Semantics
- Customization of Locale and Calendar Data
- Unicode Support

Language Support

The Oracle9*i* database enables you to store, process, and retrieve data in native languages. The languages that can be stored in an Oracle9*i* database are all languages written in scripts that are encoded by Oracle-supported character sets. Through the use of Unicode databases and datatypes, Oracle9*i* supports most contemporary languages.

Additional support is available for a subset of the languages. The Oracle9*i* database knows, for example, how to display dates using translated month names or how to sort text data according to cultural conventions.

When this manual uses the term **language support**, it refers to the additional language-dependent functionality (for example, displaying dates or sorting text), not to the ability to store text of a specific language.

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

See Also:

- Chapter 3, "Setting Up a Globalization Support Environment"
- "Languages" on page A-2 for a complete list of Oracle language names and abbreviations
- "Translated Messages" on page A-4 for a list of languages into which Oracle messages are translated

Territory Support

The Oracle9*i* database supports cultural conventions that are specific to geographical locations. The default local time format, date format, and numeric and monetary conventions depend on the local territory setting. By setting different NLS parameters, the database session can use different cultural settings. For example, you can set British pound sterling (`GBP`) as the primary currency and the Japanese yen (`JPY`) as the secondary currency for a given database session even when the territory is defined as `AMERICA`.

See Also:

- Chapter 3, "Setting Up a Globalization Support Environment"
- "Territories" on page A-5 for a list of territories that are supported by the Oracle server

Date and Time Formats

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

Time zones and daylight saving support are also available.

See Also:

- Chapter 3, "Setting Up a Globalization Support Environment"
- *Oracle9i SQL Reference*

Monetary and Numeric Formats

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

See Also: Chapter 3, "Setting Up a Globalization Support Environment"

Calendars Feature

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

See Also:

- Chapter 3, "Setting Up a Globalization Support Environment"
- "Calendar Systems" on page A-25 for a list of supported calendars

Linguistic Sorting

Oracle9*i* provides linguistic definitions for culturally accurate sorting and case conversion. The basic definition treats strings as sequences of independent characters. The extended definition recognizes pairs of characters that should be treated as special cases.

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

See Also: Chapter 4, "Linguistic Sorting"

Character Set Support

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

See Also:

- Chapter 2, "Choosing a Character Set"
- "Character Sets" on page A-6 for a list of supported character sets

Character Semantics

Oracle9*i* introduces character semantics. It is useful for defining the storage requirements for multibyte strings of varying widths in terms of characters instead of bytes.

See Also: "Length Semantics" on page 2-12

Customization of Locale and Calendar Data

You can customize locale data such as language, character set, territory, or linguistic sort using the Oracle Locale Builder.

You can customize calendars with the NLS Calendar Utility.

See Also:

- Chapter 12, "Customizing Locale Data"
- "Customizing Calendars with the NLS Calendar Utility" on page 12-17

Unicode Support

You can store Unicode characters in an Oracle9*i* database in two ways:

- You can create a Unicode database that enables you to store UTF-8 encoded characters as SQL `CHAR` datatypes.

- You can support multilingual data in specific columns by using Unicode datatypes. You can store Unicode characters into columns of the SQL `NCHAR` datatypes regardless of how the database character set has been defined. The `NCHAR` datatype has been redefined in Oracle9*i* to be an exclusively Unicode datatype.

  See Also: Chapter 5, "Supporting Multilingual Databases with Unicode"

2 Choosing a Character Set

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

- Character Set Encoding
- Length Semantics
- Choosing an Oracle Database Character Set
- Changing the Character Set After Database Creation
- Monolingual Database Scenario
- Multilingual Database Scenarios

Character Set Encoding

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

This section includes the following topics:

- What is an Encoded Character Set?
- Which Characters Are Encoded?
- What Characters Does a Character Set Support?
- How are Characters Encoded?
- Naming Convention for Oracle Character Sets

What is an Encoded Character Set?

You specify an encoded character set when you create a database. Choosing a character set determines what languages can be represented in the database. It also affects:

- How you create the database schema
- How you develop applications that process character data
- How the database works with the operating system
- Performance

A group of characters (for example, alphabetic characters, ideographs, symbols, punctuation marks, and control characters) can be encoded as a character set. An **encoded character set** assigns unique numeric codes to each character in the character repertoire. The numeric codes are called **code points** or **encoded values**. Table 2–1 shows examples of characters that have been assigned a numeric code value in the ASCII character set.

Table 2–1 Encoded Characters in the ASCII Character Set

Character	Description	Code Value
!	Exclamation Mark	21
#	Number Sign	23

Table 2–1 Encoded Characters in the ASCII Character Set (Cont.)

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

The computer industry uses many encoded character sets. Character sets differ in the following ways:

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

Oracle supports most national, international, and vendor-specific encoded character set standards.

See Also: Appendix A, "Locale Data" for a complete list of character sets that are supported by Oracle

Which Characters Are Encoded?

The characters that are encoded in a character set depend on the writing systems that are represented. A writing system can be used to represent a language or group of languages.Writing systems can be classified into two categories:

- Phonetic Writing Systems
- Ideographic Writing Systems

This section also includes the following topics:

- Punctuation, Control Characters, Numbers, and Symbols
- Writing Direction

Phonetic Writing Systems

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

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

Ideographic Writing Systems

Ideographic writing systems consist of ideographs or pictographs that represent the meaning of a word, not the sounds of a language. Chinese and Japanese are examples of ideographic writing systems that are based on tens of thousands of ideographs. Languages that use ideographic writing systems may also use a **syllabary**. Syllabaries provide a mechanism for communicating additional phonetic information. For instance, Japanese has two syllabaries: Hiragana, normally used for grammatical elements, and Katakana, normally used for foreign and onomatopoeic words.

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

Punctuation, Control Characters, Numbers, and Symbols

In addition to encoding thescript of a language, other special characters need to be encoded:

- Punctuation marks such as commas, periods, and apostrophes
- Numbers
- Special symbols such as currency symbols and math operators
- Control characters such as carriage returns and tabs

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, although 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.

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

Writing direction does not affect the encoding of a character.

What Characters 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 support more than one language. When character sets were first developed in the United States, they had a limited character repertoire. Even now there can be problems using certain characters across platforms. The following `CHAR` and `VARCHAR` characters are represented in all Oracle database character sets and can be transported to any platform:

- Uppercase and lowercase English characters A through Z and a through z
- Arabic digits 0 through 9
- The following punctuation marks: % ‘ ' ( ) * + - , . / \ : ; < > = ! _ & ~ { } | ^ ? $ # @ " [ ]
- The following control characters: space, horizontal tab, vertical tab, form feed

If you are using characters outside this set, then take care that your data is supported in the database character set that you have chosen.

Setting the `NLS_LANG` initialization parameter properly is essential to proper data conversion. The character set that is specified by the `NLS_LANG` initialization parameter should reflect the setting for the client operating system. Setting `NLS_LANG` correctly enables proper conversion from the client operating system code page to the database character set. When these settings are the same, Oracle assumes that the data being sent or received is encoded in the same character set as the database character set, so no validation or conversion is performed. This can lead to corrupt data if conversions are necessary.

During conversion from one character set to another, Oracle expects data to be encoded in the character set specified by the NLS_LANG initialization parameter. If you put other values into the string (for example, by using the CHR or CONVERT SQL functions), then the values may be corrupted when they are sent to the database because they are not converted properly. If you have configured the environment correctly and if the database character set supports the entire repertoire of character data that may be input into the database, then you do not need to change the current database character set. However, if your enterprise becomes more global and you have additional characters or new languages to support, then you may need to choose a character set with a greater character repertoire. Oracle Corporation recommends that you use Unicode databases and datatypes in these cases.

See Also:

- Chapter 5, "Supporting Multilingual Databases with Unicode"
- *Oracle9i SQL Reference* for more information about the CHR and CONVERT SQL functions
- "Displaying a Code Chart with the Oracle Locale Builder" on page 12-18

ASCII Encoding

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

Table 2–2 7-Bit ASCII 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

Table 2–2 7-Bit ASCII Character Set (Cont.)

-	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
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

Character sets have evolved to meet the needs of users around the world. New character sets have been created to support languages besides English. Typically, these new character sets support a group of related languages based on the same script. For example, the ISO 8859 character set series was created to support different European languages. Table 2–3 shows the languages that are supported by the ISO 8859 character sets.

Table 2–3 ISO 8859 Character Sets

Standard	Languages Supported
ISO 8859-1	Western European (Albanian, Basque, Breton, Catalan, Danish, Dutch, English, Faeroese, Finnish, French, German, Greenlandic, Icelandic, Irish Gaelic, Italian, Latin, 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 8859-13	Baltic Rim (English, Estonian, Finnish, Latin, Latvian, Norwegian)
ISO 8859-14	Celtic (Albanian, Basque, Breton, Catalan, Cornish, Danish, English, Galician, German, Greenlandic, Irish Gaelic, Italian, Latin, Luxemburgish, Manx Gaelic, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, Swedish, Welsh)
ISO 8859-15	Western European (Albanian, Basque, Breton, Catalan, Danish, Dutch, English, Estonian, Faroese, Finnish, French, Frisian, Galician, German, Greenlandic, Icelandic, Irish Gaelic, Italian, Latin, Luxemburgish, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, Swedish)

Character sets evolved and provided restricted multilingual support. They were restricted in the sense that they were limited to groups of languages based on similar scripts. More recently, universal character sets have been regarded as a more useful solution to multilingual support. Unicode is one such universal character set that encompasses most major scripts of the modern world. The Unicode character set supports more than 94,000 characters.

See Also: Chapter 5, "Supporting Multilingual Databases with Unicode"

How are Characters Encoded?

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

- Single-Byte Encoding Schemes
- Multibyte Encoding Schemes

Single-Byte Encoding Schemes

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

- 7-bit encoding schemes

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

- 8-bit encoding schemes

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

Figure 2–1 8-Bit Encoding Scheme

	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 because these languages use thousands of characters. These encoding schemes use either a fixed number or a variable number of bytes to represent each character.

- Fixed-width multibyte encoding schemes

  In a fixed-width multibyte encoding scheme, each character is represented by a fixed number of bytes. The number of bytes is at least two in a multibyte encoding scheme.

- Variable-width multibyte encoding schemes

  A variable-width encoding scheme uses one or more bytes to represent a single character. Some multibyte encoding schemes use certain bits to indicate the number of bytes that represents a character. For example, if two bytes is the maximum number of bytes used to represent a character, the most significant bit can be used to indicate whether that byte is a single-byte character or the first byte of a double-byte character.

- Shift-sensitive variable-width multibyte encoding schemes

Some variable-width encoding schemes use control codes to differentiate between single-byte and multibyte characters with the same code values. A shift-out code indicates that the following character is multibyte. A shift-in code indicates that the following character is single-byte. Shift-sensitive encoding schemes are used primarily on IBM platforms. Note that ISO-2022 character sets cannot be used as database character sets, but they can be used for applications such as a mail server.

Naming Convention for Oracle Character Sets

Oracle uses the following naming convention for character set names:

```
<language or region><number of bits representing a character><standard character
set name>[S | C]
```

Note: UTF8 and UTFE are exceptions to the naming convention.

The optional `S` or `C` is used to differentiate character sets that can be used only on the server (`S`) or only on the client (`C`).

Note: Use the server character set (`S`) on the Macintosh platform. The Macintosh client character sets are obsolete. On EBCDIC platforms, use the server character set (`S`) on the server and the client character set (`C`) on the client.

The following are examples of Oracle character set names.

Oracle Character Set Name	Description	Region	Number of Bits Used to Represent a Character	Standard Character Set Name
US7ASCII	U.S. 7-bit ASCII	US	7	ASCII
WE8ISO8859P1	Western European 8-bit ISO 8859 Part 1	WE (Western Europe)	8	ISO8859 Part 1
JA16SJIS	Japanese 16-bit Shifted Japanese Industrial Standard	JA	16	SJIS

Length Semantics

In single-byte character sets, the number of bytes and the number of characters in a string are the same. In multibyte character sets, a character or code unit consists of one or more bytes. Calculating the number of characters based on byte lengths can be difficult in a variable-width character set. Calculating column lengths in bytes is called **byte semantics**, while measuring column lengths in characters is called **character semantics**.

Oracle9*i* introduces character semantics. It is useful for defining the storage requirements for multibyte strings of varying widths. For example, in a Unicode database (AL32UTF8), suppose that you need to define a `VARCHAR2` column that can store up to five Chinese characters together with five English characters. Using byte semantics, this column requires 15 bytes for the Chinese characters, which are three bytes long, and 5 bytes for the English characters, which are one byte long, for a total of 20 bytes. Using character semantics, the column requires 10 characters.

The following expressions use byte semantics:

- `VARCHAR2(20 BYTE)`
- `SUBSTRB(`*`string`*`, 1, 20)`

Note the `BYTE` qualifier in the `VARCHAR2` expression and the `B` suffix in the SQL function name.

The following expressions use character semantics:

- `VARCHAR2(10 CHAR)`
- `SUBSTR(`*`string`*`, 1, 10)`

Note the `CHAR` qualifier in the `VARCHAR2` expression.

The `NLS_LENGTH_SEMANTICS` initialization parameter determines whether a new column of character datatype uses byte or character semantics. The default value of the parameter is `BYTE`. The `BYTE` and `CHAR` qualifiers shown in the `VARCHAR2` definitions should be avoided when possible because they lead to mixed-semantics databases. Instead, set `NLS_LENGTH_SEMANTICS` in the initialization parameter file and define column datatypes to use the default semantics based on the value of `NLS_LENGTH_SEMANTICS`.

Byte semantics is the default for the database character set. Character length semantics is the default and the only allowable kind of length semantics for `NCHAR` datatypes. The user cannot specify the `CHAR` or `BYTE` qualifier for `NCHAR` definitions.

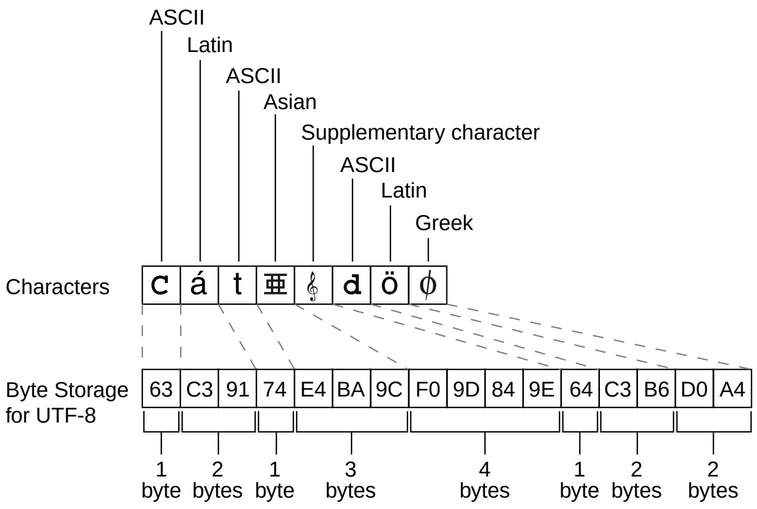
Consider the following example:

```
CREATE TABLE emp
    ( empno NUMBER(4)
    , ename NVARCHAR2(10)
    , job NVARCHAR2(9)
    , mgr NUMBER(4)
    , hiredate DATE
    , sal NUMBER(7,2)
    , deptno NUMBER(2)
    ) ;
```

When the NCHAR character set is AL16UTF16, ename can hold up to 10 Unicode code units. When the NCHAR character set is AL16UTF16, ename can hold up to 20 bytes.

Figure 2–2 shows the number of bytes needed to store different kinds of characters in the UTF-8 character set. The ASCII characters requires one byte, the Latin and Greek characters require two bytes, the Asian character requires three bytes, and the supplementary character requires four bytes of storage.

Figure 2–2 Bytes of Storage for Different Kinds of Characters



See Also:

- "SQL Functions for Different Length Semantics" on page 7-6 for more information about the SUBSTR and SUBSTRB functions
- "Length Semantics" on page 3-42 for more information about the NLS_LENGTH_SEMANTICS initialization parameter
- Chapter 5, "Supporting Multilingual Databases with Unicode" for more information about Unicode and the NCHAR datatype
- *Oracle9i SQL Reference* for more information about the SUBSTRB and SUBSTR functions and the BYTE and CHAR qualifiers for character datatypes

Choosing an Oracle Database Character Set

Oracle uses the database character set for:

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

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

Note: CLOB data is encoded as UCS-2 if the database character set is multibyte. If the database character set is single-byte, then CLOB data is stored in the database character set.

After the database is created, you cannot change the character sets, with some exceptions, without re-creating the database.

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

- What languages does the database need to support now?
- What languages will the database need to support in the future?

- Is the character set available on the operating system?
- What character sets are used on clients?
- How well does the application handle the character set?
- What are the performance implications of the character set?
- What are the restrictions associated with the character set?

The Oracle character sets are listed in Appendix A, "Locale Data". They are named according to the languages and regions in which they are used. Some character sets that are named for a region are also listed explicitly by language.

If you want to see the characters that are included in a character set, then:

- Check national, international, or vendor product documentation or standards documents
- Use Oracle Locale Builder

This section contains the following topics:

- Current and Future Language Requirements
- Client Operating System and Application Compatibility
- Character Set Conversion Between Clients and the Server
- Performance Implications of Choosing a Database Character Set
- Restrictions on Database Character Sets
- Choosing a National Character Set
- Summary of Supported Datatypes

See Also:

- "UCS-2 Encoding" on page 5-4
- "Choosing a National Character Set" on page 2-18
- "Changing the Character Set After Database Creation" on page 2-20
- Appendix A, "Locale Data"
- Chapter 12, "Customizing Locale Data"

Current and Future Language Requirements

Several character sets may meet your current language requirements. Consider future language requirements when you choose a database character set. If you expect to support additional languages in the future, then choose a character set that supports those languages to prevent the need to migrate to a different character set later.

Client Operating System and Application Compatibility

The database character set is independent of the operating system because Oracle has its own globalization architecture. For example, on an English Windows operating system, you can create and run a database with a Japanese character set. However, when the client operating system accesses the database, the client operating system must be able to support the database character set with appropriate fonts and input methods. For example, you cannot insert or retrieve Japanese data on the English Windows operating system without first installing a Japanese font and input method. Another way to insert and retrieve Japanese data is to use a Japanese operating system remotely to access the database server.

Character Set Conversion Between Clients and the Server

If you choose a database character set that is different from the character set on the client operating system, then the Oracle database can convert the operating system character set to the database character set. Character set conversion has the following disadvantages:

- Increased overhead
- Potential data loss

Character set conversions can sometimes cause data loss. For example, if you are converting from character set A to character set B, then the destination character set B must have the same character set repertoire as A. Any characters that are not available in character set B are converted to a replacement character. The replacement character is often specified as a question mark or as a linguistically related character. For example, ä (a with an umlaut) may 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 several times before the data reaches the client. The database character set should always be a superset or equivalent of the native character set of the client's operating system.

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

See Also: Chapter 10, "Character Set Migration"

Performance Implications of Choosing a Database Character Set

For best performance, choose a character set that avoids character set conversion and uses the most efficient encoding for the languages desired. Single-byte character sets result in better performance than multibyte character sets, and they also are the most efficient in terms of space requirements. However, single-byte character sets limit how many languages you can support.

Restrictions on Database Character Sets

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

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

Restrictions on Character Sets Used to Express Names

Table 2–4 lists the restrictions on the character sets that can be used to express names.

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

Name	Single-Byte	Variable Width	Comments
column names	Yes	Yes	-
schema objects	Yes	Yes	-
comments	Yes	Yes	-

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

Name	Single-Byte	Variable Width	Comments
database link names	Yes	No	-
database names	Yes	No	-
file names (datafile, log file, control file, initialization parameter file)	Yes	No	-
instance names	Yes	No	-
directory names	Yes	No	-
keywords	Yes	No	Can be expressed in English ASCII or EBCDIC characters only
Recovery Manager file names	Yes	No	-
rollback segment names	Yes	No	The ROLLBACK_SEGMENTS parameter does not support NLS
stored script names	Yes	Yes	-
tablespace names	Yes	No	-

For a list of supported string formats and character sets, including LOB data (LOB, BLOB, CLOB, and NCLOB), see Table 2–6 on page 2-19.

Choosing a National Character Set

A **national character set** is an alternate character set that enables you to store Unicode character data in a database that does not have a Unicode database character set. Other reasons for choosing a national character set are:

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

SQL NCHAR, NVARCHAR2, and NCLOB datatypes have been redefined to support Unicode data only. You can store the data in either UTF-8 or UTF-16 encodings.

See Also: Chapter 5, "Supporting Multilingual Databases with Unicode"

Summary of Supported Datatypes

Table 2–5 lists the datatypes that are supported for different encoding schemes.

Table 2–5 SQL Datatypes Supported for Encoding Schemes

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

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

Table 2–6 lists the SQL datatypes that are supported for abstract datatypes.

Table 2–6 Abstract Datatype Support for SQL Datatypes

Abstract Datatype	CHAR	NCHAR	BLOB	CLOB	NCLOB
Object	Yes	Yes	Yes	Yes	Yes
Collection	Yes	Yes	Yes	Yes	Yes

You can create an abstract datatype with the `NCHAR` attribute as follows:

```
SQL> CREATE TYPE tp1 AS OBJECT (a NCHAR(10));
Type created.
SQL> CREATE TABLE t1 (a tp1);
Table created.
```

See Also: *Oracle9i Application Developer's Guide - Object-Relational Features* for more information about objects and collections

Changing the Character Set After Database Creation

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

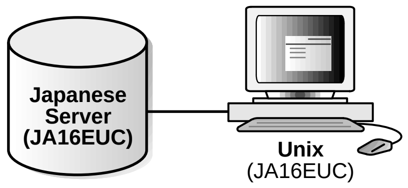
See Also:

- Chapter 10, "Character Set Migration"
- *Oracle9i Database Migration* for more information about exporting and importing data
- *Oracle9i SQL Reference* for more information about the ALTER DATABASE CHARACTER SET statement

Monolingual Database Scenario

The simplest example of a database configuration is a client and a server that run in the same language environment and use the same character set. This monolingual scenario has the advantage of fast response because the overhead associated with character set conversion is avoided. Figure 2–3 shows a database server and a client that use the same character set.

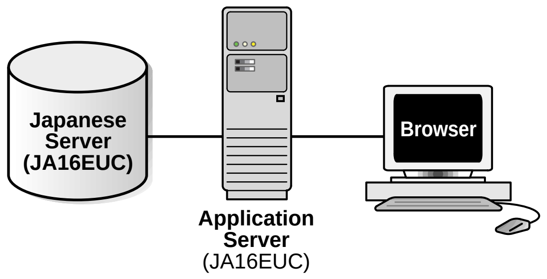
Figure 2–3 Monolingual Database Scenario



The Japanese client and the server both use the JA16EUC character set.

You can also use a multitier architecture. Figure 2–4 shows an application server between the database server and the client. The application server and the database server use the same character set in a monolingual scenario.

Figure 2–4 Multitier Monolingual Database Scenario



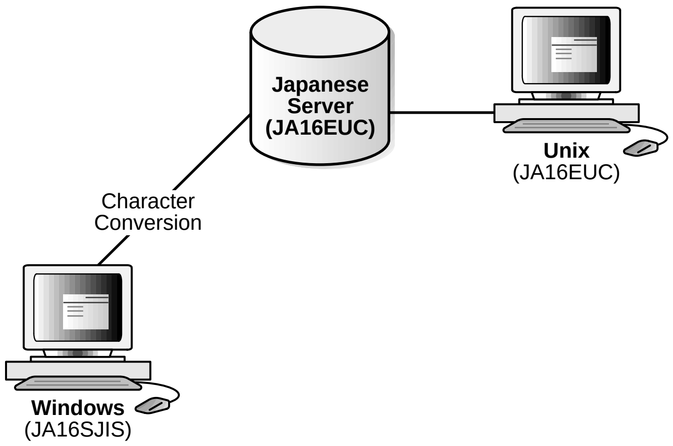
The server, the application server, and the client use the JA16EUC character set.

Character Set Conversion in a Monolingual Scenario

Character set conversion may be required in a client/server environment if a client application resides on a different platform than the server and if the platforms do not use the same character encoding schemes. Character data passed between client and server must be converted between the two encoding schemes. Character conversion occurs automatically and transparently via Oracle Net.

You can convert between any two character sets. Figure 2–5 shows a server and one client with the JA16EUC Japanese character set. The other client uses the JA16SJIS Japanese character set.

Figure 2–5 Character Set Conversion

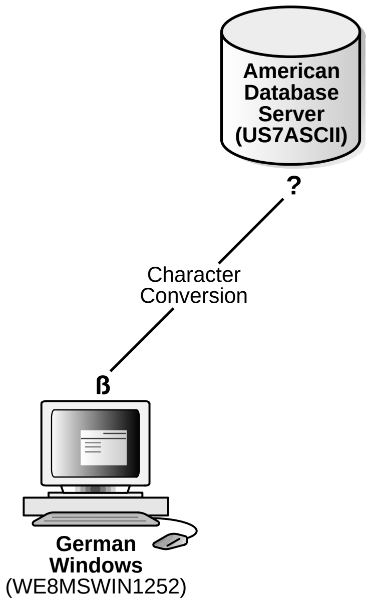


When a target character set does not contain all of the characters in the source data, replacement characters are used. If, for example, a server uses US7ASCII and a German client uses WE8ISO8859P1, the German character ß is replaced with ? and ä is replaced with a.

Replacement characters may be defined for specific characters as part of a character set definition. When a specific replacement character is not defined, a default replacement character is used. To avoid the use of replacement characters when converting from a client character set to a database character set, the server character set should be a superset of all the client character sets.

Figure 2–6 shows that data loss occurs when the database character set does not include all of the characters in the client character set.

Figure 2–6 Data Loss During Character Conversion



The database character set is US7ASCII. The client's character set is WE8MSWIN1252, and the language used by the client is German. When the client inserts a string that contains ß, the database replaces ß with ?, resulting in lost data.

If German data is expected to be stored on the server, then a database character set that supports German characters should be used for both the server and the client to avoid data loss and conversion overhead.

When one of the character sets is a variable-width multibyte character set, conversion can introduce noticeable overhead. Carefully evaluate your situation and choose character sets to avoid conversion as much as possible.

Multilingual Database Scenarios

Multilingual support can be restricted or unrestricted. This section contains the following topics:

- Restricted Multilingual Support
- Unrestricted Multilingual Support

Restricted Multilingual Support

Some character sets support multiple languages because they have related writing systems or scripts. For example, the WE8ISO8859P1 Oracle character set supports the following Western European languages:

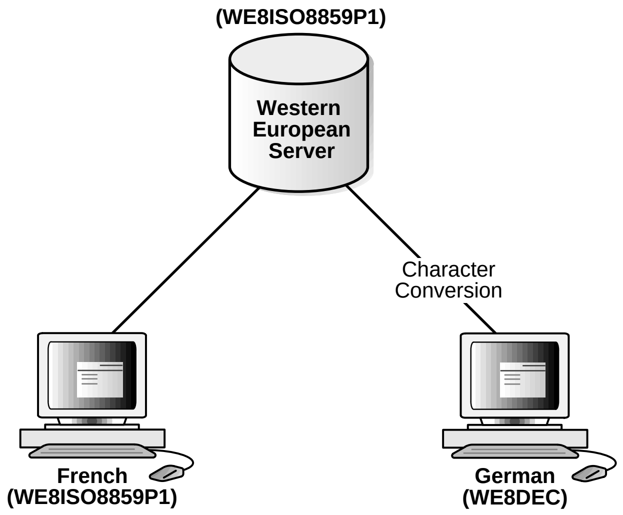
Catalan
Danish
Dutch
English
Finnish
French
German
Icelandic
Italian
Norwegian
Portuguese
Spanish
Swedish

These languages all use a Latin-based writing script.

When you use a character set that supports a group of languages, your database has **restricted multilingual support**.

Figure 2–7 shows a Western European server that used the WE8ISO8850P1 Oracle character set, a French client that uses the same character set as the server, and a German client that uses the WE8DEC character set. The German client requires character conversion because it is using a different character set than the server.

Figure 2–7 Restricted Multilingual Support



Unrestricted Multilingual Support

If you need unrestricted multilingual support, use a universal character set such as Unicode for the server database character set. Unicode has two major encoding schemes:

- UTF-16: Each character is either 2 or 4 bytes long.
- UTF-8: Each character takes 1 to 4 bytes to store.

The Oracle9*i* database provides support for UTF-8 as a database character set and both UTF-8 and UTF-16 as national character sets.

Character set conversion between a UTF-8 database and any single-byte character set introduces very little overhead.

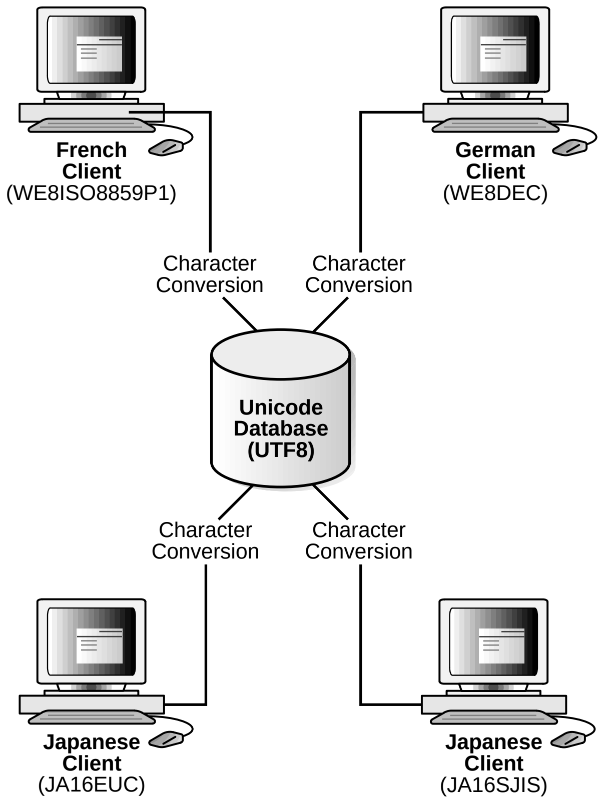
Conversion between UTF-8 and any multibyte character set has some overhead. There is no data loss from conversion with the following exceptions:

- Some multibyte character sets do not support user-defined characters during character set conversion to and from UTF-8.
- Some Unicode characters are mapped to more than character in another character set. For example, one Unicode character is mapped to three characters

in the JA16SJIS character set. This means that a round-trip conversion may not result in the original JA16SJIS character.

Figure 2–8 shows a server that uses the AL32UTF8 Oracle character set that is based on the Unicode UTF-8 character set.

Figure 2–8 Unrestricted Multilingual Support Scenario in a Client/Server Configuration



There are four clients:

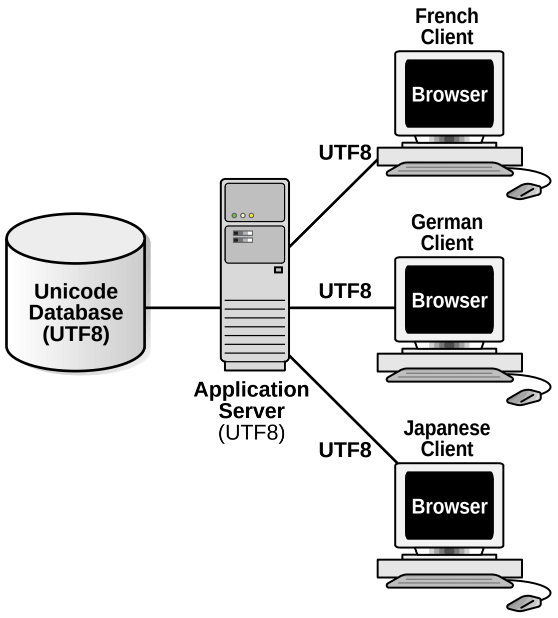
- A French client that uses the WE8ISO8859P1 Oracle character set
- A German client that uses the WE8DEC character set
- A Japanese client that uses the JA16EUC character set

- A Japanese client that used the JA16SJIS character set

Character conversion takes place between each client and the server, but there is no data loss because AL32UTF8 is a universal character set. If the German client tries to retrieve data from one of the Japanese clients, all of the Japanese characters in the data are lost during the character set conversion.

Figure 2–9 shows a Unicode solution for a multitier configuration.

Figure 2–9 Multitier Unrestricted Multilingual Support Scenario in a Multitier Configuration



The database, the application server, and each client use the AL32UTF8 character set. This eliminates the need for character conversion even though the clients are French, German, and Japanese.

See Also: Chapter 5, "Supporting Multilingual Databases with Unicode"

3 Setting Up a Globalization Support Environment

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

Setting NLS Parameters

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

- As initialization parameters on the server

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

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

- As environment variables on the client

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

  ```
  % setenv NLS_SORT FRENCH
  ```

- With the `ALTER SESSION` statement

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

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

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

- In SQL functions

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

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

  > **See Also:** *Oracle9i SQL Reference* for more information about SQL functions, including the `TO_CHAR` function

Table 3–1 shows the precedence order of the different methods of setting NLS parameters. Higher priority settings override lower priority settings. For example, a default value has the lowest priority and can be overridden by any other method. Another example is that setting an NLS parameter within a SQL function overrides all other methods of setting NLS parameters.

Table 3–1 Methods of Setting NLS Parameters and Their Priorities

Priority	Method
1 (highest)	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

Table 3–2 lists the NLS parameters available with the Oracle server.

Table 3–2 NLS Parameters

Parameter	Description	Default	Scope:
			I = Initialization Parameter File
			E = Environment Variable
			A = ALTER SESSION

NLS_CALENDAR	Calendar system	Gregorian	I, E, A
NLS_COMP	SQL, PL/SQL operator comparison	BINARY	I, E, A
NLS_CREDIT	Credit accounting symbol	Derived from NLS_TERRITORY	E
NLS_CURRENCY	Local currency symbol	Derived from NLS_TERRITORY	I, E, A
NLS_DATE_FORMAT	Date format	Derived from NLS_TERRITORY	I, E, A
NLS_DATE_LANGUAGE	Language for day and month names	Derived from NLS_LANGUAGE	I, E, A
NLS_DEBIT	Debit accounting symbol	Derived from NLS_TERRITORY	E
NLS_ISO_CURRENCY	ISO international currency symbol	Derived from NLS_TERRITORY	I, E, A

Table 3–2 NLS Parameters (Cont.)

Parameter	Description	Default	Scope:
			I = Initialization Parameter File
			E = Environment Variable
			A = ALTER SESSION

NLS_LANG **See Also:** "Choosing a Locale with the NLS_LANG Environment Variable" on page 3-4	Language, territory, character set	AMERICAN_AMERICA.US7ASCII	E
NLS_LANGUAGE	Language	Derived from NLS_LANG	I, A
NLS_LENGTH_SEMANTICS	How strings are treated	BYTE	I, A
NLS_LIST_SEPARATOR	Character that separates items in a list	Derived from NLS_TERRITORY	E
NLS_MONETARY_CHARACTERS	Monetary symbol for dollar and cents (or their equivalents)	Derived from NLS_TERRITORY	E
NLS_NCHAR_CONV_EXCP	Reports data loss during a character type conversion	FALSE	I, A
NLS_NUMERIC_CHARACTERS	Decimal character and group separator	Derived from NLS_TERRITORY	I, E, A
NLS_SORT	Character sort sequence	Derived from NLS_LANGUAGE	I, E, A
NLS_TERRITORY	Territory	Derived from NLS_LANG	I, A
NLS_TIMESTAMP_FORMAT	Timestamp	Derived from NLS_TERRITORY	I, E, A
NLS_TIMESTAMP_TZ_FORMAT	Timestamp with time zone	Derived from NLS_TERRITORY	I, E, A
NLS_DUAL_CURRENCY	Dual currency symbol	Derived from NLS_TERRITORY	I, E, A

Choosing a Locale with the NLS_LANG Environment Variable

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

The `NLS_LANG` parameter has three components: language, territory, and character set. Specify it in the following format, including the punctuation:

```
NLS_LANG = language_territory.charset
```

For example, if the Oracle Installer does not populate `NLS_LANG`, then its value is `AMERICAN_AMERICA.US7ASCII`. The language is `AMERICAN`, the territory is `AMERICA`, and the character set is `US7ASCII`.

Each component of the `NLS_LANG` parameter controls the operation of a subset of globalization support features:

- *language*

  Specifies conventions such as the language used for Oracle messages, sorting, day names, and month names. Each supported language has a unique name; for example, `AMERICAN`, `FRENCH`, or `GERMAN`. The language argument specifies default values for the territory and character set arguments. If the language is not specified, then the value defaults to `AMERICAN`.

- *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 the territory is not specified, then the value is derived from the language value.

- *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`, `WE8MSWIN1252`, or `JA16EUC`. Each language has a default character set associated with it.

Note: All components of the `NLS_LANG` definition are optional; any item left out will default. If you specify territory or character set, you *must* include the preceding delimiter [underscore (_) for territory, period (.) for character set]. 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.WE8MSWIN1252

NLS_LANG = FRENCH_CANADA.WE8DEC

NLS_LANG = JAPANESE_JAPAN.JA16EUC
```

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

```
NLS_LANG = JAPANESE_JAPAN.WE8DEC
```

Because the WE8DEC character set does not support any Japanese characters, you cannot store Japanese data if you use this definition for `NLS_LANG`.

The rest of this section includes the following topics:

- Specifying the Value of NLS_LANG
- Overriding Language and Territory Specifications

> **See Also:** Appendix A, "Locale Data" for a complete list of supported languages, territories, and character sets

Specifying the Value of NLS_LANG

Set `NLS_LANG` as an environment variable at the command line. For example, in the UNIX operating system, specify the value of `NLS_LANG` by entering a statement similar to the following:

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

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

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

Example 3–1 Setting NLS_LANG to American_America.WE8ISO8859P1

Set `NLS_LANG` so that the language is `AMERICAN`, the territory is `AMERICA`, and the Oracle character set is `WE8ISO8859P1`:

```
% setenv NLS_LANG American_America.WE8ISO8859P1
```

Enter a SELECT statement:

```
SQL> SELECT last_name, hire_date, ROUND(salary/8,2) salary FROM employees;
```

You should see results similar to the following:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
Sciarra                   30-SEP-97      962.5
Urman                     07-MAR-98        975
Popp                      07-DEC-99      862.5
```

Example 3–2 Setting NLS_LANG to French_France.WE8ISO8859P1

Set NLS_LANG so that the language is FRENCH, the territory is FRANCE, and the Oracle character set is WE8ISO8859P1:

```
% setenv NLS_LANG French_France.WE8ISO8859P1
```

Then the query shown in Example 3–1 returns the following output:

```
LAST_NAME                 HIRE_DAT     SALARY
------------------------- -------- ----------
Sciarra                   30/09/97      962,5
Urman                     07/03/98        975
Popp                      07/12/99      862,5
```

Note that the date format and the number format have changed. The numbers have not changed, because the underlying data is the same.

Overriding Language and Territory Specifications

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

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

If the NLS_LANG parameter is not set, then 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 the NLS environment dynamically during the session. To do so, you can use the ALTER SESSION statement to change NLS_LANGUAGE, NLS_TERRITORY, and other NLS parameters.

Note: You cannot modify the setting for the client character set with 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.

See Also:

- "Overriding Default Values for NLS_LANGUAGE and NLS_TERRITORY During a Session" on page 3-16
- *Oracle9i SQL Reference*

Should the NLS_LANG Setting Match the Database Character Set?

The NLS_LANG character set should reflect the setting of the operating system client. For example, if the database character set is UTF8 and the client has a Windows operating system, you should not set UTF8 as the client character set because there are no UTF8 WIN32 clients. Instead the NLS_LANG setting should reflect the code page of the client.

NLS_LANG is set as a local environment variable on UNIX platforms.

NLS_LANG is set in the registry on Windows platforms. For example, on an English Windows client, the code page is WE8MSWIN1252. An appropriate setting for NLS_LANG is AMERICAN_AMERICA.WE8MSWIN1252.

Setting NLS_LANG correctly allows proper conversion from the client operating system code page to the database character set. When these settings are the same, Oracle assumes that the data being sent or received is encoded in the same character set as the database character set, so no validation or conversion is performed. This can lead to corrupt data if the client code page and the database character set are different and conversions are necessary.

See Also: *Oracle9i Database Installation Guide for Windows* for more information about commonly used values of the NLS_LANG parameter in Windows

NLS Database Parameters

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

You can find the values for NLS parameters by using:

- NLS Data Dictionary Views
- NLS Dynamic Performance Views
- OCINlsGetInfo() Function

NLS Data Dictionary Views

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

- NLS_SESSION_PARAMETERS shows the NLS parameters and their values for the session that is querying the view. It does not show information about the character set.
- NLS_INSTANCE_PARAMETERS shows the current NLS instance parameters that have been explicitly set and the values of the NLS instance parameters.
- NLS_DATABASE_PARAMETERS shows the values of the NLS parameters that were used when the database was created.

NLS Dynamic Performance Views

Applications can check the following NLS dynamic performance views:

- V$NLS_VALID_VALUES lists values for the following NLS parameters: NLS_LANGUAGE, NLS_SORT, NLS_TERRITORY, NLS_CHARACTERSET
- V$NLS_PARAMETERS shows current values of the following NLS parameters: NLS_CALENDAR, NLS_CHARACTERSET, NLS_CURRENCY, NLS_DATE_FORMAT, NLS_DATE_LANGUAGE, NLS_ISO_CURRENCY, NLS_LANGUAGE, NLS_

NUMERIC_CHARACTERS, NLS_SORT, NLS_TERRITORY, NLS_NCHAR_CHARACTERSET, NLS_COMP, NLS_LENGTH_SEMANTICS, NLS_NCHAR_CONV_EXP, NLS_TIMESTAMP_FORMAT, NLS_TIMESTAMP_TZ_FORMAT, NLS_TIME_FORMAT, NLS_TIME_TZ_FORMAT

See Also: *Oracle9i Database Reference*

OCINlsGetInfo() Function

User applications can query client NLS settings with the OCINlsGetInfo() function.

See Also: Chapter 8, "OCI Programming in a Global Environment" for the description of OCINlsGetInfo()

Language and Territory Parameters

This section contains information about the following parameters:

- NLS_LANGUAGE
- NLS_TERRITORY

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. (A.M., P.M., A.D., and B.C. are valid only if NLS_LANGUAGE is set to AMERICAN.)
- Default sorting sequence for character data when ORDER BY is specified. (GROUP BY uses a binary sort unless ORDER BY is specified.)

- Writing direction
- Affirmative and negative response strings (for example, `YES` and `NO`)

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

```
NLS_LANGUAGE = FRENCH
```

When the language is French, the server message

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

appears as

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

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

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

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

Messages are stored in these files in one specific character set, depending on the language and the operating system. If this character set is different from the database character set, then message text is automatically converted to the database character set. If necessary, it will then be converted to the client character set if the client character set 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 specific to the operating system. You can alter the `NLS_LANGUAGE` parameter by changing its value in the initialization parameter file and then restarting the instance.

> **See Also:** Your operating system-specific Oracle documentation for more information about the default value of `NLS_LANGUAGE`

All messages and text should be in the same language. For example, when you run an Oracle Developer application, the messages and boilerplate text that you see originate from three sources:

- Messages from the server
- Messages and boilerplate text generated by Oracle Forms
- Messages and boilerplate text generated by the application

NLS determines the language used for the first two kinds of text. The application is responsible for the language used in its messages and boilerplate text.

The following examples show behavior that results from setting `NLS_LANGUAGE` to different values.

Example 3–3 NLS_LANGUAGE=ITALIAN

Use the `ALTER SESSION` statement to set `NLS_LANGUAGE` to Italian:

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

Enter a `SELECT` statement:

```
SQL> SELECT last_name, hire_date, ROUND(salary/8,2) salary FROM employees;
```

You should see results similar to the following:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
Sciarra                   30-SET-97      962.5
Urman                     07-MAR-98        975
Popp                      07-DIC-99      862.5
```

Note that the month name abbreviations are in Italian.

> **See Also:** "Overriding Default Values for NLS_LANGUAGE and NLS_TERRITORY During a Session" on page 3-16 for more information about using the `ALTER SESSION` statement

Example 3–4 NLS_LANGUAGE=GERMAN

Use the `ALTER SESSION` statement to change the language to German:

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

Enter the same `SELECT` statement:

```
SQL> SELECT last_name, hire_date, ROUND(salary/8,2) salary FROM employees;
```

You should see results similar to the following:

```
LAST_NAME                 HIRE_DATE     SALARY
```

```
------------------------- --------- ----------
Sciarra                   30-SEP-97      962.5
Urman                     07-MÄR-98        975
Popp                      07-DEZ-99      862.5
```

Note that the language of the month abbreviations has changed.

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
- First day of the week
- Credit and debit symbols
- ISO week flag
- List separator

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

```
NLS_TERRITORY = FRANCE
```

When the territory is `FRANCE`, numbers are formatted using a comma as the decimal character.

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

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

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

To change the territory to France during a session, issue the following ALTER SESSION statement:

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

The following examples show behavior that results from different settings of NLS_TERRITORY and NLS_LANGUAGE.

Example 3–5 NLS_LANGUAGE=AMERICAN, NLS_TERRITORY=AMERICA

Enter the following SELECT statement:

```
SQL> SELECT TO_CHAR(salary,'L99G999D99') salary FROM employees;
```

When NLS_TERRITORY is set to AMERICA and NLS_LANGUAGE is set to AMERICAN, results similar to the following should appear:

```
SALARY
--------------------
$24,000.00
$17,000.00
$17,000.00
```

Example 3–6 NLS_LANGUAGE=AMERICAN, NLS_TERRITORY=GERMANY

Use an ALTER SESSION statement to change the territory to Germany:

```
ALTER SESSION SET NLS_TERRITORY = Germany;
Session altered.
```

Enter the same SELECT statement as before:

```
SQL> SELECT TO_CHAR(salary,'L99G999D99') salary FROM employees;
```

You should see results similar to the following:

```
SALARY
--------------------
€24.000,00
€17.000,00
€17.000,00
```

Note that the currency symbol has changed from $ to €. The numbers have not changed because the underlying data is the same.

> **See Also:** "Overriding Default Values for NLS_LANGUAGE and NLS_TERRITORY During a Session" on page 3-16 for more information about using the `ALTER SESSION` statement

Example 3–7 NLS_LANGUAGE=GERMAN, NLS_TERRITORY=GERMANY

Use an `ALTER SESSION` statement to change the language to German:

```
ALTER SESSION SET NLS_LANGUAGE = German;
Sitzung wurde geändert.
```

Note that the server message now appears in German.

Enter the same `SELECT` statement as before:

```
SQL> SELECT TO_CHAR(salary,'L99G999D99') salary FROM employees;
```

You should see the same results as in Example 3–6:

```
SALARY
--------------------
€24.000,00
€17.000,00
€17.000,00
```

Example 3–8 NLS_LANGUAGE=GERMAN, NLS_TERRITORY=AMERICA

Use an `ALTER SESSION` statement to change the territory to America:

```
ALTER SESSION SET NLS_TERRITORY = America;
Sitzung wurde geändert.
```

Enter the same `SELECT` statement as in the other examples:

```
SQL> SELECT TO_CHAR(salary,'L99G999D99') salary FROM employees;
```

You should see output similar to the following:

```
SALARY
```

```
--------------------
$24.000,00
$17.000,00
$17.000,00
```

Note that the currency symbol changed from € to $ because the territory changed from Germany to America.

Overriding Default Values for NLS_LANGUAGE and NLS_TERRITORY During a Session

Default values for NLS_LANGUAGE and NLS_TERRITORY can be overridden during a session by using the ALTER SESSION statement.

Example 3–9 NLS_LANG=ITALIAN_ITALY.WE8DEC

Set the NLS_LANG environment variable so that the language is Italian, the territory is Italy, and the character set is WE8DEC:

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

Enter a SELECT statement:

```
SQL> SELECT last_name, hire_date, ROUND(salary/8,2) salary FROM employees;
```

You should see output similar to the following:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
Sciarra                   30-SET-97      962,5
Urman                     07-MAR-98        975
Popp                      07-DIC-99      862,5
```

Note the language of the month abbreviations and the decimal character.

Example 3–10 Change Language, Date Format, and Decimal Character

Use ALTER SESSION statements to change the language, the date format, and the decimal character:

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

Session wurde geändert.

SQL> ALTER SESSION SET NLS_DATE_FORMAT='DD.MON.YY';
```

```
Session wurde geändert.

SQL> ALTER SESSION SET NLS_NUMERIC_CHARACTERS='.,';

Session wurde geändert.
```

Enter the SELECT statement shown in Example 3–9:

```
SQL> SELECT last_name, hire_date, ROUND(salary/8,2) salary FROM employees;
```

You should see output similar to the following:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
Sciarra                   30.SEP.97      962.5
Urman                     07.MÄR.98        975
Popp                      07.DEZ.99      862.5
```

Note the language of the month abbreviations, the date format, and the decimal character.

The behavior of the NLS_LANG environment variable implicitly determines the language environment of the database for each session. When a session connects to a database, an ALTER SESSION statement is automatically executed 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, then the changes are propagated to all instances to which that user session is connected.

Date and Time Parameters

Oracle enables you to control the display of date and time. This section contains the following topics:

- Date Formats
- Time Formats

Date Formats

Different date formats are shown in Table 3–3.

Table 3–3 Date Formats

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

This section includes the following parameters:

- NLS_DATE_FORMAT
- NLS_DATE_LANGUAGE

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

The NLS_DATE_FORMAT parameter defines the default date format to use with the TO_CHAR and TO_DATE functions. The NLS_TERRITORY parameter determines the default value of NLS_DATE_FORMAT. The value of NLS_DATE_FORMAT can be any valid date format mask. 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'
```

Example 3–11 Setting the Date Format to Display Roman Numerals

To set the default date format to display Roman numerals for the month, include the following line in the initialization parameter file:

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

Enter the following `SELECT` statement:

```
SELECT TO_CHAR(SYSDATE) currdate FROM dual;
```

You should see the following output if today's date is February 12, 1997:

```
CURRDATE
---------
12 II 1997
```

The value of `NLS_DATE_FORMAT` 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 (month, day, and year), two one-byte strings (the two slashes), and the two-byte terminator for the format mask. The format for the value of `NLS_DATE_FORMAT` 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 parameter file and then restarting the instance
- Using an `ALTER SESSION SET NLS_DATE_FORMAT` statement

See Also: *Oracle9i SQL ReferenceOracle9i SQL Reference* for more information about date format elements and the `ALTER SESSION` statement

If a table or index is partitioned on a date column, and if the date format specified by `NLS_DATE_FORMAT` does not specify the first two digits of the year, then you must use the `TO_DATE` function with a 4-character format mask for the year.

For example:

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

> **See Also:** *Oracle9i SQL Reference* for more information about partitioning tables and indexes and using `TO_DATE`

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

The `NLS_DATE_LANGUAGE` parameter specifies the language for the day and month names produced by the `TO_CHAR` and `TO_DATE` functions. `NLS_DATE_LANGUAGE` overrides the language that is specified implicitly by `NLS_LANGUAGE`. `NLS_DATE_LANGUAGE` has the same syntax as the `NLS_LANGUAGE` parameter, and all supported languages are valid values.

`NLS_DATE_LANGUAGE` also determines the language used for:

- Month and day abbreviations returned by the `TO_CHAR` and `TO_DATE` functions
- Month and day abbreviations used by the default date format (`NLS_DATE_FORMAT`)
- Abbreviations for AM, PM, AD, and BC

Example 3–12 NLS_DATE_LANGUAGE=FRENCH, Month and Day Names

Set the date language to French:

```
ALTER SESSIONS SET NLS_DATE_LANGUAGE = FRENCH
```

Enter a `SELECT` statement:

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

You should see output similar to the following:

```
TO_CHAR(SYSDATE,'DAY:DDMONTHYYYY')
------------------------------------------------------------
```

```
Vendredi:07 Décembre  2001
```

When numbers are spelled in words using the `TO_CHAR` function, the English spelling is always used. For example, enter the following `SELECT` statement:

```
SQL> SELECT TO_CHAR(TO_DATE('12-Oct-2001'),'Day: ddspth Month') FROM dual;
```

You should see output similar to the following:

```
TO_CHAR(TO_DATE('12-OCT-2001'),'DAY:DDSPTHMONTH')
--------------------------------------------------------------------
Vendredi: twelfth Octobre
```

_Example 3–13   NLS_DATE_LANGUAGE=FRENCH, Month and Day Abbreviations_

Month and day abbreviations are determined by `NLS_DATE_LANGUAGE`. Enter the following `SELECT` statement:

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

You should see output similar to the following:

```
TO_CHAR(SYSDATE,'DY:DDMO
------------------------
Ve:07 Dec 2001
```

_Example 3–14   NLS_DATE_LANGUAGE=FRENCH, Default Date Format_

The default date format uses the month abbreviations determined by `NLS_DATE_LANGUAGE`. For example, if the default date format is `DD-MON-YYYY`, then insert a date as follows:

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

See Also: *Oracle9i SQL Reference*

Time Formats

Different time formats are shown in Table 3–4.

Table 3–4   Time Formats

Country	Description	Example
Estonia	hh24:mi:ss	13:50:23
Germany	hh24:mi:ss	13:50:23

Table 3–4 Time Formats (Cont.)

Country	Description	Example
Japan	hh24:mi:ss	13:50:23
UK	hh24:mi:ss	13:50:23
US	hh:mi:ssxff am	1:50:23.555 PM

This section contains information about the following parameters:

- NLS_TIMESTAMP_FORMAT
- NLS_TIMESTAMP_TZ_FORMAT

NLS_TIMESTAMP_FORMAT

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

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

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

Example 3–15 Timestamp Format

```
SQL> SELECT TO_TIMESTAMP('11-nov-2000 01:00:00.336', 'dd-mon-yyyy hh:mi:ss.ff')
    FROM dual;
```

You should see output similar to the following:

```
TO_TIMESTAMP('11-NOV-200001:00:00.336','DD-MON-YYYYHH:MI:SS.FF')
---------------------------------------------------------------------------
11-NOV-00 01:00:00.336000000
```

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

You can also alter the value of NLS_TIMESTAMP_FORMAT by:

- Changing its value in the initialization parameter file and then restarting the instance
- Using the ALTER SESSION SET NLS_TIMESTAMP_FORMAT statement

> **See Also:** *Oracle9i SQL Reference* for more information about the TO_TIMESTAMP function and the ALTER SESSION statement

NLS_TIMESTAMP_TZ_FORMAT

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

NLS_TIMESTAMP_TZ_FORMAT defines the default format for the timestamp with time zone. It is used with the TO_CHAR and TO_TIMESTAMP_TZ functions.

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

Example 3–16 Setting NLS_TIMESTAMP_TZ_FORMAT

The format value must be surrounded by quotation marks. For example:

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

The following example of the TO_TIMESTAMP_TZ function uses the format value that was specified for NLS_TIMESTAMP_TZ_FORMAT:

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

You should see output similar to the following:

```
TO_TIMESTAMP_TZ('2000-08-20,05:00:00.44AMERICA/LOS_ANGELES','YYYY-MM-DDHH:M
---------------------------------------------------------------------------
20-AOU-00 05:00:00.440000000 AMERICA/LOS_ANGELES
```

You can change the value of NLS_TIMESTAMP_TZ_FORMAT by:

- Changing its value in the initialization parameter file and then restarting the instance
- Using the ALTER SESSION statement.

> **See Also:** *Oracle9i SQL Reference* for more information about the TO_TIMESTAMP_TZ function and the ALTER SESSION statement

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

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

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

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

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

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

The database time zone is relevant only for TIMESTAMP WITH LOCAL TIME ZONE columns. Oracle normalizes all TIMESTAMP WITH LOCAL TIME ZONE data to the time zone of the database when the data is stored on disk. If you do not specify the SET TIME_ZONE clause, then Oracle uses the time zone of the operating system of the server. If the operating system's time zone is not a valid Oracle time zone, then the rdatabase time zone defaults to UTC. Oracle's time zone information is derived from the public domain time zone data available at ftp://elsie.nci.nih.gov/pub/. Oracle's time zone information may not reflect the most recent time zone data available from this site.

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

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

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

```
SELECT dbtimezone FROM dual;
```

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

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

- Operating system local time zone

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

- Database time zone

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

- An absolute time difference from UTC

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

- Time zone for a named region

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

You can use the `ORA_SDTZ` environment variable to set the default client session time zone. This variable takes input like `DB_TZ`, `OS_TZ`, time zone region, or numerical time zone offset. If `ORA_SDTZ` is set to `DB_TZ`, then the session time zone will be the same as the database time zone. If it is set to `OS_TZ`, then the session time zone will be same as the operating system's time zone. If `ORA_SDTZ` is set to an invalid Oracle time zone, then Oracle uses the operating system's time zone as default session time zone. If the operating system's time zone is not a valid Oracle time zone, then the session time zone defaults to UTC. To find out the time zone of a user session, use the `SESSIONTIMEZONE` function as shown in the following example:

```
SELECT sessiontimezone FROM dual;

SESSIONTIMEZONE
---------------
         -08:00
```

See Also: "Customizing Time Zone Data" on page 12-17

Calendar Definitions

This section includes the following topics:

- Calendar Formats
- NLS_CALENDAR

Calendar Formats

The following calendar information is stored for each territory:

- 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, as shown in Table 3–5.

Table 3–5 German Calendar Example: March 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	-	-	-	-	-

The first day of the week is determined by the `NLS_TERRITORY` parameter.

See Also: "NLS_TERRITORY" on page 3-13

First Calendar Week of the Year

Some countries use week numbers for scheduling, planning, and bookkeeping. Oracle supports this convention. In the ISO standard, the week number can be different from the week number of the calendar year. For example, 1st Jan 1988 is in ISO week number 53 of 1987. An ISO week always starts on a Monday and ends on a Sunday.

- If January 1 falls on a Friday, Saturday, or Sunday, then the ISO week that includes 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 ISO 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, Oracle provides the IW date format element. It returns the ISO week number.

Table 3–6 shows an example in which January 1 occurs in a week that has four or more days in the first calendar week of the year. The week containing January 1 is the first ISO week of 1998.

Table 3–6 First ISO Week of the Year: Example 1, January 1998

Mo	Tu	We	Th	Fr	Sa	Su	ISO Week
-	-	-	1	2	3	4	First ISO week of 1998
5	6	7	8	9	10	11	Second ISO week of 1998
12	13	14	15	16	17	18	Third ISO week of 1998
19	20	21	22	23	24	25	Fourth ISO week of 1998
26	27	28	29	30	31	-	Fifth ISO week of 1998

Table 3–7 shows an example in which January 1 occurs in a week that has three or fewer days in the first calendar week of the year. The week containing January 1 is the 53rd ISO week of 1998, and the following week is the first ISO week of 1999.

Table 3–7 First ISO Week of the Year: Example 2, January 1999

Mo	Tu	We	Th	Fr	Sa	Su	ISO Week
-	-	-	-	1	2	3	Fifty-third ISO week of 1998
4	5	6	7	8	9	10	First ISO week of 1999

Table 3–7 First ISO Week of the Year: Example 2, January 1999 (Cont.)

Mo	Tu	We	Th	Fr	Sa	Su	ISO Week
11	12	13	14	15	16	17	Second ISO week of 1999
18	19	20	21	22	23	24	Third ISO week of 1999
25	26	27	28	29	30	31	Fourth ISO week of 1999

The first calendar week of the year is determined by the `NLS_TERRITORY` parameter.

See Also: "NLS_TERRITORY" on page 3-13

Number of Days and Months in a Year

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

- 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 31 days for each of the first six months. The next five months have 30 days each. The last month has either 29 days or 30 days (leap year).
- 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

The calendar system is specified by the `NLS_CALENDAR` parameter.

See Also: "NLS_CALENDAR" on page 3-29

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

> **See Also:** Appendix A, "Locale Data" for a list of calendar systems, their default date formats, and the character sets in which dates are displayed

Example 3–17 NLS_CALENDAR='Japanese Imperial"

Set NLS_CALENDAR to Japanese Imperial:

```
SQL> ALTER SESSIONS SET NLS_CALENDAR='English Hijrah';
```

Enter a SELECT statement to display SYSDATE:

```
SELECT SYSDATE FROM dual;
```

You should see output similar to the following:

```
SYSDATE
--------------------
24 Ramadan     1422
```

Numeric Parameters

This section includes the following topics:

- Numeric Formats
- NLS_NUMERIC_CHARACTERS

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, applications must be able to display numeric information in the format expected at the client site.

Examples of numeric formats are shown in Table 3–8.

Table 3–8 Examples of Numeric Formats

Country	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

Numeric formats are derived from the setting of the `NLS_TERRITORY` parameter, but they can be overridden by the `NLS_NUMERIC_CHARACTERS` parameter.

See Also: "NLS_TERRITORY" on page 3-13

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 group separator. The group separator is the character that separates integer groups to show thousands and millions, for example. The group separator is the character returned by the G number format mask. The decimal character separates the integer and decimal parts of a number. Setting `NLS_NUMERIC_CHARACTERS` overrides the values derived from the setting of `NLS_TERRITORY`.

Any character can be the decimal or group separator. The two characters specified must be single-byte, and the 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 (>). Either character can be a space.

The characters are specified in the following format:

```
NLS_NUMERIC_CHARACTERS = "decimal_character group_separator"
```

Example 3–18   Setting NLS_NUMERIC_CHARACTERS

To set the decimal character to a comma and the grouping separator to a period, define `NLS_NUMERIC_CHARACTERS` as follows:

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS = ",.";
```

Both characters are single-byte and are different from each other.

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.

Enter a `SELECT` statement:

```
SELECT TO_CHAR(4000, '9G999D99') FROM dual;
```

You should see output similar to the following:

```
TO_CHAR(4
---------
 4.000,00
```

You can change the default value of `NLS_NUMERIC_CHARACTERS` by:

- Changing the value of `NLS_NUMERIC_CHARACTERS` in the initialization parameter file and then restart the instance

- Using the `ALTER SESSION` statement to change the parameter's value during a session

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

Monetary Parameters

This section includes the following topics:

- Currency Formats
- NLS_CURRENCY
- NLS_ISO_CURRENCY
- NLS_DUAL_CURRENCY
- NLS_MONETARY_CHARACTERS
- NLS_CREDIT
- NLS_DEBIT

Currency Formats

Different currency formats are used throughout the world. Some typical ones are shown in Table 3–9.

Table 3–9 Currency Format Examples

Country	Example
Estonia	1 234,56 kr
Germany	1.234,56€
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

NLS_CURRENCY specifies the character string returned by the L number format mask, the local currency symbol. Setting NLS_CURRENCY overrides the setting defined implicitly by NLS_TERRITORY.

Example 3–19 Displaying the Local Currency Symbol

Connect to the sample schema order entry schema:

```
SQL> connect oe/oe
Connected.
```

Enter a SELECT statement similar to the following:

```
SQL> SELECT TO_CHAR(order_total, 'L099G999D99') "total" FROM orders
   WHERE order_id > 2450;
```

You should see output similar to the following:

```
total
---------------------
        $078,279.60
        $006,653.40
        $014,087.50
        $010,474.60
        $012,589.00
        $000,129.00
        $003,878.40
        $021,586.20
```

You can change the default value of NLS_CURRENCY by:

- Changing its value in the initialization parameter file and then restarting the instance
- Using an ALTER SESSION statement

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

`NLS_ISO_CURRENCY` specifies the character string returned by the `C` number format mask, the ISO currency symbol. Setting `NLS_ISO_CURRENCY` overrides the value defined implicitly by `NLS_TERRITORY`.

Local currency symbols can be ambiguous. For example, a dollar sign ($) can refer to US dollars or Australian dollars. ISO specifications define unique currency symbols for specific territories or countries. For example, the ISO currency symbol for the US dollar is USD. For the Australian dollar, it is AUD.

More ISO currency symbols are shown in Table 3–10.

Table 3–10 ISO Currency Examples

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

`NLS_ISO_CURRENCY` has the same syntax as the `NLS_TERRITORY` parameter, and all supported territories are valid values.

Example 3–20 Setting NLS_ISO_CURRENCY

This example assumes that you are connected as `oe/oe` in the sample schema.

To specify the ISO currency symbol for France, set `NLS_ISO_CURRENCY` as follows:

```
ALTER SESSION SET NLS_ISO_CURRENCY = FRANCE;
```

Enter a `SELECT` statement:

```
SQL> SELECT TO_CHAR(order_total, 'C099G999D99') "TOTAL" FROM orders
    WHERE customer_id = 146;
```

You should see output similar to the following:

```
TOTAL
------------------
EUR017,848.20
EUR027,455.30
EUR029,249.10
EUR013,824.00
EUR000,086.00
```

You can change the default value of `NLS_ISO_CURRENCY` by:

- Changing its value in the initialization parameter file and then restarting the instance
- Using an `ALTER SESSION` statement

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

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

Use `NLS_DUAL_CURRENCY` to override the default dual currency symbol defined implicitly by `NLS_TERRITORY`.

`NLS_DUAL_CURRENCY` was introduced to support the euro currency symbol during the euro transition period. Table 3–11 lists the character sets that support the euro symbol.

Table 3–11 Character Sets that Support the Euro Symbol

Character Set Name	Description	Code Value of the Euro Symbol
D8EBCDIC1141	EBCDIC Code Page 1141 8-bit Austrian German	0x9F
DK8EBCDIC1142	EBCDIC Code Page 1142 8-bit Danish	0x5A
S8EBCDIC1143	EBCDIC Code Page 1143 8-bit Swedish	0x5A
I8EBCDIC1144	EBCDIC Code Page 1144 8-bit Italian	0x9F
F8EBCDIC1147	EBCDIC Code Page 1147 8-bit French	0x9F
WE8PC858	IBM-PC Code Page 858 8-bit West European	0xDF
WE8ISO8859P15	ISO 8859-15 West European	0xA4
EE8MSWIN1250	MS Windows Code Page 1250 8-bit East European	0x80
CL8MSWIN1251	MS Windows Code Page 1251 8-bit Latin / Cyrillic	0x88
WE8MSWIN1252	MS Windows Code Page 1252 8-bit West European	0x80
EL8MSWIN1253	MS Windows Code Page 1253 8-bit Latin / Greek	0x80
WE8EBCDIC1047E	Latin 1 / Open Systems 1047	0x9F
WE8EBCDIC1140	EBCDIC Code Page 1140 8-bit West European	0x9F
WE8EBCDIC1140C	EBCDIC Code Page 1140 Client 8-bit West European	0x9F
WE8EBCDIC1145	EBCDIC Code Page 1145 8-bit West European	0x9F
WE8EBCDIC1146	EBCDIC Code Page 1146 8-bit West European	0x9F
WE8EBCDIC1148	EBCDIC Code Page 1148 8-bit West European	0x9F
WE8EBCDIC1148C	EBCDIC Code Page 1148 Client 8-bit West European	0x9F
EL8ISO8859P7	ISO 8859-7 Latin / Greek	0xA4
IW8MSWIN1255	MS Windows Code Page 1255 8-bit Latin / Hebrew	0x80
AR8MSWIN1256	MS Windows Code Page 1256 8-Bit Latin / Arabic	0x80
TR8MSWIN1254	MS Windows Code Page 1254 8-bit Turkish	0x80
BLT8MSWIN1257	MS Windows Code Page 1257 Baltic	0x80
VN8MSWIN1258	MS Windows Code Page 1258 8-bit Vietnamese	0x80
TH8TISASCII	Thai Industrial 620-2533 - ASCII 8-bit	0x80
AL32UTF8	Unicode 3.1 UTF-8 Universal character set	E282AC
UTF8	Unicode 3.0 UTF-8 Universal character set	E282AC
AL16UTF16	Unicode 3.1 UTF-16 Universal character set	20AC

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

Character Set Name	Description	Code Value of the Euro Symbol
UTFE	UTF-EBCDIC encoding of Unicode 3.0	CA4653
ZHT16HKSCS	MS Windows Code Page 950 with Hong Kong Supplementary Character Set	0xA3E1
ZHS32GB18030	GB18030-2000	0xA2E3
WE8BS2000E	Siemens EBCDIC.DF.04 8-bit West European	0x9F

Oracle Support for the Euro

The members of the European Monetary Union (EMU) now use the euro as their currency as of January 1, 2002. Setting NLS_TERRITORY to correspond to a country in the EMU (Austria, Belgium, Finland, France, Germany, Greece, Ireland, Italy, Luxembourg, the Netherlands, Portugal, and Spain) results in the default values for NLS_CURRENCY and NLS_DUAL_CURRENCY being set to EUR.

During the transition period (1999 through 2001), Oracle support for the euro was provided in Oracle8*i* and later as follows:

- NLS_CURRENCY was defined as the primary currency of the country
- NLS_ISO_CURRENCY was defined as the ISO currency code of a given territory
- NLS_DUAL_CURRENCY was defined as the secondary currency symbol (usually the euro) for a given territory

Beginning with Oracle9*i* release 2 (9.2), the value of NLS_ISO_CURRENCY results in the ISO currency symbol being set to EUR for EMU member countries. For example, suppose NLS_ISO_CURRENCY is set to FRANCE. Enter the following SELECT statement:

```
SELECT TO_CHAR(TOTAL, ’C099G999D99’) "TOTAL" FROM orders WHERE customer_id=585;
```

You should see output similar to the following:

```
TOTAL
-------
EUR12.673,49
```

Customers who must retain their obsolete local currency symbol can override the default for NLS_DUAL_CURRENCY or NLS_CURRENCY by defining them as parameters in the initialization file on the server and as environment variables on the client.

Note: NLS_LANG must also be set on the client for NLS_CURRENCY or NLS_DUAL_CURRENCY to take effect.

It is not possible to override the ISO currency symbol that results from the value of NLS_ISO_CURRENCY.

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 character that separates groups of numbers in monetary expressions. For example, when the territory is America, the thousands separator is a comma, and the decimal separator is a period.

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. For example, a space is a valid value of NLS_CREDIT.

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`. For example, a minus sign (-) is a valid value of `NLS_DEBIT`.

This parameter can be specified only in the client environment.

It can be retrieved through the `OCIGetNlsInfo()` function.

Linguistic Sort Parameters

You can choose how to sort data by using linguistic sort parameters.

This section includes the following topics:

- NLS_SORT
- NLS_COMP
- NLS_LIST_SEPARATOR

See Also: Chapter 4, "Linguistic Sorting"

NLS_SORT

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

`NLS_SORT` specifies the type of sort for character data. It overrides the value that is defined implicitly by `NLS_LANGUAGE`.

The syntax of `NLS_SORT` is:

```
NLS_SORT = BINARY | sort_name
```

`BINARY` specifies a binary sort. *`sort_name`* specifies a linguistic sort sequence.

Example 3–21 Setting NLS_SORT

To specify the linguistic sort sequence called German, set NLS_SORT 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 that uses the same name. Oracle offers two kinds of linguistic sorts: monolingual and multilingual. In addition monolingual sorts can be extended to handle special cases. Extended monolingual sorts usually sort characters differently than the ASCII values of the characters. For example, ch and ll are treated as only one character in XSPANISH, the extended Spanish sort. In other words, the SPANISH sort uses modern Spanish collation rules, while XSPANISH uses traditional Spanish sorting rules.

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

When NLS_SORT is set to a linguistic sort, a sort is needed to satisfy the ORDER BY clause if there is no linguistic index for the linguistic sort specified by NLS_SORT.

If a linguistic index exists for the linguistic sort specified by NLS_SORT, the optimizer can, in some cases, satisfy the ORDER BY clause without doing a sort by choosing an index scan.

You can alter the default value of NLS_SORT by:

- Changing its value in the initialization parameter file and then restarting the instance
- Using an ALTER SESSION statement

See Also:

- "Multilingual Linguistic Sorts" on page 4-4
- *Oracle9i SQL Reference* for more information about the ALTER SESSION statement
- Appendix A, "Locale Data" for a complete list of linguistic sort 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 NLS_COMP to avoid the cumbersome process of using NLS_SORT in SQL statements. Normally, comparison in the WHERE clause and in PL/SQL blocks is binary. To use linguistic comparison, you must use the NLSSORT SQL function. Sometimes this can be tedious, especially when the linguistic sort has already been specified in the NLS_SORT session parameter. You can use NLS_COMP to indicate that the comparisons must be linguistic according to the NLS_SORT session parameter. Do this by altering the session:

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

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

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

When NLS_COMP is set to ANSI, a linguistic index improves the performance of the linguistic comparison.

To enable a linguistic index, use the following syntax:

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

See Also: "Using Linguistic Indexes" on page 4-12

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. Its default value is derived from the value of NLS_TERRITORY.

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 Conversion Parameter

This section includes the following topic:

- NLS_NCHAR_CONV_EXCP

NLS_NCHAR_CONV_EXCP

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

`NLS_NCHAR_CONV_EXCP` determines whether an error is reported when there is data loss during an implicit or explicit character type conversion. The default value results in no error being reported.

See Also: Chapter 10, "Character Set Migration" for more information about data loss during character set conversion

Length Semantics

This section includes the following topic:

- NLS_LENGTH_SEMANTICS

NLS_LENGTH_SEMANTICS

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

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

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

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

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

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

See Also:

- "Length Semantics" on page 2-12
- *Oracle9i Database Concepts* for more information about length semantics

4 Linguistic Sorting

This chapter explains how characters are sorted in an Oracle environment. It contains the following topics:

Overview of Oracle's Sorting Capabilities

Different languages have different sort orders. In addition, different cultures or countries that use the same alphabets may sort words differently. For example, in Danish, Æ is after Z, while Y and Ü are considered to be variants of the same letter.

Sort order can be case-sensitive or case-insensitive. **Case** refers to the condition of being uppercase or lowercase. For example, in a Latin alphabet, A is the uppercase glyph for a, the lowercase glyph.

Sort order can ignore or consider diacritics. A **diacritic** is a mark near or through a character or combination of characters that indicates a different sound than the sound of the character without the diacritic. For example, the cedilla (,) in façade is a diacritic. It changes the sound of c.

Sort order can be phonetic or it can be based on the appearance of the character. For example, sort order can be based on the number of strokes in East Asian ideographs. Another common sorting issue is combining letters into a single character. For example, in traditional Spanish, ch is a distinct character that comes after c, which means that the correct order is: cerveza, colorado, cheremoya. This means that the letter c cannot be sorted until Oracle has checked whether the next letter is an h.

Oracle provides the following types of sorts:

- Binary sort
- Monolingual linguistic sort
- Multilingual linguistic sort

It can achieve a linguistically correct sort for a single language as well as a sort based on the multilingual ISO standard (ISO-14651), which is designed to handle many languages at the same time.

Using Binary Sorts

One way to sort character data is based on the numeric values of the characters defined by the character encoding scheme. This is called a **binary sort**. Binary sorts are the fastest type of sort. They produce reasonable results for the English alphabet because the ASCII and EBCDIC standards define the letters A to Z in ascending numeric value.

Note: In the ASCII standard, all uppercase letters appear before any lowercase letters. In the EBCDIC standard, the opposite is true: all lowercase letters appear before any uppercase letters.

When characters used in other languages are present, a binary sort usually does not produce reasonable results. For example, an ascending ORDER BY query returns the character strings ABC, ABZ, BCD, ÄBC, when Ä has a higher numeric value than B in the character encoding scheme. A binary sort is not usually linguistically meaningful for Asian languages that use ideographic characters.

Using Linguistic Sorts

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

Oracle offers two kinds of linguistic sorts: monolingual and multilingual.

This section includes the following topics:

- Monolingual Linguistic Sorts
- Multilingual Linguistic Sorts
- Multilingual Sorting Levels
- Linguistic Sort Examples

Monolingual Linguistic Sorts

Oracle compares character strings in two steps for monolingual sorts. The first step compares the major value of the entire string from a table of major values. Usually, letters with the same appearance have the same major value. The second step compares the minor value from a table of minor values. The major and minor values are defined by Oracle. Oracle defines letters with diacritic and case differences as having the same major value but different minor values.

Each major table entry contains the **Unicode code point** and major value for a character. The Unicode code point is a 16-bit binary value that represents a character.

Table 4–1 illustrates sample values for sorting `a`, `A`, `ä`, `Ä`, and `b`.

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

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

See Also: "Overview of Unicode" on page 5-2

Multilingual Linguistic Sorts

Oracle9*i* provides multilingual linguistic sorts so that you can sort data in more than one language in one sort. This is useful for regions or languages that have complex sorting rules and for multilingual databases. Oracle9*i* supports all of the sort orders defined by previous releases.

For Asian language data or multilingual data, Oracle provides a sorting mechanism based on the ISO 14651 standard and the Unicode 3.1 standard. Chinese characters are ordered by the number of strokes, PinYin, or radicals.

In addition, multilingual sorts can handle canonical equivalence and supplementary characters. **Canonical equivalence** is a basic equivalence between characters or sequences of characters. For example, `ç` is equivalent to the combination of `c` and `,`. **Supplementary characters** are user-defined characters or predefined characters in Unicode 3.1 that require two code points within a specific code range. You can define up to 1.1 million code points in one multilingual sort.

For example, Oracle9*i* supports a monolingual French sort (`FRENCH`), but you can specify a multilingual French sort (`FRENCH_M`). `_M` represents the ISO 14651 standard for multilingual sorting. The sorting order is based on the `GENERIC_M` sorting order and can sort diacritical marks from right to left. Oracle Corporation recommends using a multilingual linguistic sort if the tables contain multilingual data. If the tables contain only French, then a monolingual French sort may have better performance because it uses less memory. It uses less memory because fewer characters are defined in a monolingual French sort than in a multilingual French sort. There is a tradeoff between the scope and the performance of a sort.

See Also:

- "Canonical Equivalence" on page 4-10
- "Supplementary Characters" on page 5-3

Multilingual Sorting Levels

Oracle evaluates multilingual sorts at three levels of precision:

- Primary Level Sorts
- Secondary Level Sorts
- Tertiary Level Sorts

Primary Level Sorts

A primary level sort distinguishes between **base characters**, such as the difference between characters `a` and `b`. It is up to individual locales to define if `a` is before `b`, `b` is before `a`, or they are equal. The binary representation of the characters is completely irrelevant. If a character is an ignorable character, then it is assigned a primary level **order** (or weight) of zero, which means it is ignored at the primary level. Characters that are ignorable on other levels are given an order of zero at those levels.

For example, at the primary level, all variations of `bat` come before all variations of `bet`. The variations of `bat` can appear in any order, and the variations of `bet` can appear in any order:

```
Bat
bat
BAT
BET
Bet
bet
```

See Also: "Ignorable Characters" on page 4-9

Secondary Level Sorts

A secondary level sort distinguishes between base characters (the primary level sort) before distinguishing between diacritics on a given base character. For example, the character Ä differs from the character A only because it has a diacritic. Thus, Ä and A are the same on the primary level because they have the same base character (A) but differ on the secondary level.

The following list has been sorted on the primary level (`resume` comes before `resumes`) and on the secondary level (strings without diacritics come before strings with diacritics):

```
resume
résumé
Résumé
Resumes
resumes
résumés
```

Tertiary Level Sorts

A tertiary level sort distinguishes between base characters (primary level sort), diacritics (secondary level sort), and case (upper case and lower case). It can also include special characters such as `+`, `-`, and `*`.

The following are examples of tertiary level sorts:

- Characters `a` and `A` are equal on the primary and secondary levels but different on the tertiary level because they have different cases.
- Characters `ä` and `A` are equal on the primary level and different on the secondary and tertiary levels.
- The primary and secondary level orders for the dash character `-` is 0. That is, it is ignored on the primary and secondary levels. If a dash is compared with another character whose primary level order is nonzero, for example, `u`, then no result for the primary level is available because `u` is not compared with anything. In this case, Oracle finds a difference between `-` and `u` only at the tertiary level.

The following list has been sorted on the primary level (`resume` comes before `resumes`) and on the secondary level (strings without diacritics come before strings with diacritics) and on the tertiary level (lower case comes before upper case):

```
resume
Resume
résumé
Résumé
resumes
résumés
Resumes
Résumés
```

Linguistic Sort Examples

The examples in this section demonstrate a binary sort, a monolingual sort, and a multilingual sort. To prepare for the examples, create and populate a table called `test`. Enter the following statements:

```
SQL> CREATE TABLE test (name VARCHAR2(20));
SQL> INSERT INTO test VALUES('Diet');
SQL> INSERT INTO test VALUES('À voir');
SQL> INSERT INTO test VALUES('Freizeit');
```

Example 4–1 Binary Sort

The `ORDER BY` clause uses a binary sort.

```
SQL> SELECT * FROM test ORDER BY name;
```

You should see the following output:

```
Diet
Freizeit
À voir
```

Note that a binary sort results in `À voir` being at the end of the list.

Example 4–2 Monolingual German Sort

Use the `NLSSORT` function with the `NLS_SORT` parameter set to `german` to obtain a German sort.

```
SQL> SELECT * FROM test ORDER BY NLSSORT(name, 'NLS_SORT=german');
```

You should see the following output:

```
À voir
Diet
Freizeit
```

Note that `À voir` is at the beginning of the list in a German sort.

Example 4–3 Comparing a Monolingual German Sort to a Multilingual Sort

Insert the character string shown in Figure 4–1 into `test`. It is a `D` with a crossbar followed by `ñ`.

Figure 4–1 Character String

Đñ

Perform a monolingual German sort by using the `NLSSORT` function with the `NLS_SORT` parameter set to `german`.

```
SQL> SELECT * FROM test ORDER BY NLSSORT(name, 'NLS_SORT=german');
```

The output from the German sort shows the new character string last in the list of entries because the characters are not recognized in a German sort.

Perform a multilingual sort by entering the following statement:

```
SQL> SELECT * FROM test ORDER BY NLSSORT(name, 'NLS_SORT=generic_m');
```

The output shows the new character string after `Diet`, following ISO sorting rules.

See Also:

- "The NLSSORT Function" on page 7-10
- "NLS_SORT" on page 3-39 for more information about setting and changing the `NLS_SORT` parameter

Linguistic Sort Features

This section contains information about different features that a linguistic sort may have:

- Base Letters
- Ignorable Characters
- Contracting Characters
- Expanding Characters
- Context-Sensitive Characters
- Canonical Equivalence
- Reverse Secondary Sorting
- Character Rearrangement for Thai and Laotian Characters
- Special Letters
- Special Combination Letters

- Special Uppercase Letters
- Special Lowercase Letters

You can customize linguistic sorts to include the desired characteristics.

See Also: Chapter 12, "Customizing Locale Data"

Base Letters

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

See Also: *Oracle Text Reference*

Ignorable Characters

Some characters can be ignored in a linguistic sort. These characters are called **ignorable characters**. There are two kinds of ignorable characters: diacritics and punctuation.

Examples of ignorable diacritics are:

- `^`, so that `rôle` is treated the same as `role`
- The umlaut, so that `naïve` is treated the same as `naive`

And example of an ignorable punctuation character is the dash character `-`. If it is ignored, then `multi-lingual` can be treated that same as `multilingual` and `e-mail` can be treated the same as `email.`

Contracting Characters

Sorting elements usually consist of a single character, but in some locales, two or more characters in a character string must be considered as a single sorting element during sorting. For example, in traditional Spanish, the string `ch` is composed of two characters. These characters are called **contracting characters** in multilingual linguistic sorting and **special combination letters** in monolingual linguistic sorting.

Do not confuse a **composed character** with a contracting character. A composed character like `á` can be decomposed into `a` and `'`, each with their own encoding. The difference between a composed character and a contracting character is that a composed character can be displayed as a single character on a terminal, while a contracting character is used only for sorting, and its component characters must be rendered separately.

Expanding Characters

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

Context-Sensitive Characters

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

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

Canonical Equivalence

One Unicode code point may be equivalent to a sequence of base character code points plus diacritic code points, regardless of the locale. This is called the Unicode canonical equivalence. For example, ä equals its base letter a and an umlaut. A linguistic flag, CANONICAL_EQUIVALENCE=TRUE, indicates that all canonical equivalence rules defined in Unicode 3.1 need to be applied. You can change this flag to FALSE to speed up the comparison and ordering functions if all the data is in its composed form.

See Also: "Creating a New Linguistic Sort with the Oracle Locale Builder" on page 12-35 for more information about setting the canonical equivalence flag

Reverse Secondary Sorting

In French, sorting strings of characters with diacritics first compares base characters from left to right, but compares characters with diacritics from right to left. For example, by default, a character with a diacritic is placed after its unmarked variant. Thus `Èdit` comes before `Edít` in a French sort. They are equal on the primary level, and the secondary order is determined by examining characters with diacritics from right to left. Individual locales can request that the characters with diacritics be sorted with the right-to-left rule. Set the `REVERSE_SECONDARY` linguistic flag to `TRUE` to enable reverse secondary sorting.

See Also: "Creating a New Linguistic Sort with the Oracle Locale Builder" on page 12-35 for more information about setting the reverse secondary flag

Character Rearrangement for Thai and Laotian Characters

In Thai and Lao, some characters must first change places with the following character before sorting. Normally, these types of character are symbols representing vowel sounds, and the next character is a consonant. Consonants and vowels must change places before sorting. Set the `SWAP_WITH_NEXT` linguistic flag for all characters that must change places before sorting.

See Also: "Creating a New Linguistic Sort with the Oracle Locale Builder" on page 12-35 for more information about setting the `SWAP_WITH_NEXT` flag

Special Letters

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

See Also: "Expanding Characters" on page 4-10

Special Combination Letters

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

See Also: "Contracting Characters" on page 4-9

Special Uppercase Letters

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

These case conversions are handled by the NLS_UPPER, NLS_LOWER, and NLS_INITCAP SQL functions, according to the conventions established by the linguistic sort sequence. The UPPER, LOWER, and INITCAP SQL functions cannot handle these special characters.

The NLS_UPPER SQL function returns all uppercase characters from the same character set as the lowercase string. The following example shows the result of the NLS_UPPER function when NLS_SORT is set to XGERMAN:

```
SELECT NLS_UPPER ('große') "Uppercase" FROM DUAL;

Upper
-----
GROSSE
```

See Also: *Oracle9i SQL Reference*

Special Lowercase Letters

Oracle supports special lowercase letters. One uppercase letter may map to multiple lowercase letters. An example is the Turkish uppercase I becoming a small, dotless i: ı.

Using Linguistic Indexes

Linguistic sorting is language-specific and requires more data processing than binary sorting. Using a binary sort for ASCII is accurate and fast because the binary codes for ASCII characters reflect their linguistic order. When data in multiple languages is stored in the database, you may want applications to sort the data returned from a SELECT...ORDER BY statement according to different sort sequences depending on the language. You can accomplish this without sacrificing performance by using linguistic indexes. Although a linguistic index for a column slows down inserts and updates, it greatly improves the performance of linguistic sorting with the ORDER BY clause.

You can create a function-based index that uses languages other than English. The index does not change the linguistic sort order determined by NLS_SORT. The index simply improves the performance. The following statement creates an index based on a German sort:

```
CREATE TABLE my_table(name VARCHAR(20) NOT NULL)
/*NOT NULL ensures that the index will be used */
CREATE INDEX nls_index ON my_table (NLSSORT(name, 'NLS_SORT = German'));
```

After the index has been created, enter a SELECT statement similar to the following:

```
SELECT * FROM my_table ORDER BY name;
```

It returns the result much faster than the same SELECT statement without an index.

The rest of this section contains the following topics:

- Linguistic Indexes for Multiple Languages
- Requirements for Using Linguistic Indexes

See Also:

- *Oracle9i Database Concepts*
- *Oracle9i SQL Reference* for more information about function-based indexes

Linguistic Indexes for Multiple Languages

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

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

  ```
  CREATE INDEX french_index ON employees (NLSSORT(employee_id, 'NLS_
  SORT=FRENCH'));
  CREATE INDEX german_index ON employees (NLSSORT(employee_id, 'NLS_
  SORT=GERMAN'));
  ```

  Oracle chooses the index based on the NLS_SORT session parameter or the arguments of the NLSSORT function specified in the ORDER BY clause. For example, if the NLS_SORT session parameter is set to FRENCH, Oracle uses french_index. When it is set to GERMAN, Oracle uses german_index.

- Build a single linguistic index for all languages. This requires a language column (LANG_COL in "Example: Setting Up a French Linguistic Index" on page 4-15) to be used as a parameter of the NLSSORT function. The language column contains NLS_LANGUAGE values for the data in the column on which the index is built. The following example builds a single linguistic index for multiple languages. With this index, the rows with the same values for NLS_LANGUAGE are sorted together.

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

  Queries choose an index based on the argument of the NLSSORT function specified in the ORDER BY clause.

- Build a single linguistic index for all languages using one of the multilingual linguistic sorts such as GENERIC_M or FRENCH_M. These indexes sort characters according to the rules defined in ISO 14651. For example:

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

  See Also: "Multilingual Linguistic Sorts" on page 4-4 for more information about Unicode sorts

Requirements for Using Linguistic Indexes

The following are requirements for using linguistic indexes:

- Set QUERY_REWRITE_ENABLED to TRUE
- Set NLS_COMP to ANSI
- Set NLS_SORT Appropriately
- Use the Cost-Based Optimizer With the Optimizer Mode Set to FIRST_ROWS

This section also includes:

- Example: Setting Up a French Linguistic Index

Set QUERY_REWRITE_ENABLED to TRUE

The QUERY_REWRITE_ENABLED initialization parameter must be set to TRUE. This is required for all function-based indexes. You can use an ALTER SESSION statement to set QUERY_REWRITE_ENABLED to TRUE. For example:

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

See Also: *Oracle9i Database Reference* for more information about the QUERY_REWRITE_ENABLED initialization parameter

Set NLS_COMP to ANSI

The NLS_COMP parameter should be set to ANSI. There are several ways to set NLS_COMP. For example:

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

See Also: "NLS_COMP" on page 3-41

Set NLS_SORT Appropriately

The NLS_SORT parameter should indicate the linguistic definition you want to use for the linguistic sort. If you want a French linguistic sort order, NLS_SORT should be set to FRENCH. If you want a German linguistic sort order, NLS_SORT should be set to GERMAN.

There are several ways to set NLS_SORT. You should set NLS_SORT as a client environment variable so that you can use the same SQL statements for all languages. Different linguistic indexes can be used when NLS_SORT is set in the client environment.

See Also: "NLS_SORT" on page 3-39

Use the Cost-Based Optimizer With the Optimizer Mode Set to FIRST_ROWS

Use the cost-based optimizer with the optimizer mode set to FIRST_ROWS, because linguistic indexes are not recognized by the rule-based optimizer. The following is an example of setting the optimizer mode:

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

See Also: *Oracle9i Database Performance Guide and Reference* for more information about the cost-based optimizer

Example: Setting Up a French Linguistic Index

The following example shows how to set up a French linguistic index. You may want to set NLS_SORT as a client environment variable instead of using the ALTER SESSION statement.

```
ALTER SESSION SET QUERY_REWRITE_ENABLED=TRUE;
ALTER SESSION SET NLS_COMP = ANSI;
ALTER SESSION SET NLS_SORT='FRENCH';
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS;
CREATE INDEX test_idx ON test(NLSSORT(col, 'NLS_SORT=FRENCH'));
SELECT * FROM test ORDER BY col;
SELECT * FROM test WHERE col > 'JJJ';
```

Improving Case-Insensitive Searches with a Function-Based Index

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

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

Performing a Generic Base Letter Search

You can perform a search that ignores case and diacritics. Enter the following statements:

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

Then enter a statement similar to the following:

```
SELECT * FROM emp WHERE ename='miller';
```

This statement can now return names that include the following:

```
Miller
MILLER
Millér
```

Note that this is not a linguistic search; that is, it is not based on a specific language. It uses the base letters only.

5 Supporting Multilingual Databases with Unicode

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

Overview of Unicode

Dealing with many different languages in the same application or database has been complicated and difficult for a long time. To overcome the limitations of existing character encodings, several organizations began working on the creation of a global character set in the late 1980s. The need for this became even greater with the development of the World Wide Web in the mid-1990s. The Internet has changed how companies do business, with an emphasis on the global market that has made a universal character set a major requirement. A global character set needs to fulfill the following conditions:

- Contain all major living scripts
- Support legacy data and implementations
- Be simple enough that a single implementation of an application is sufficient for worldwide use

A global character set should also have the following capabilities:

- Support multilingual users and organizations
- Conform to international standards
- Enable worldwide interchange of data

This global character set exists, is in wide use, and is called Unicode.

What is Unicode?

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

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

Oracle Corporation started supporting Unicode as a database character set in Oracle7. In Oracle9*i*, Unicode support has been expanded. Oracle9*i* supports Unicode 3.1.

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

This section contains the following topics:

- Supplementary Characters
- Unicode Encodings
- Oracle's Support for Unicode

Supplementary Characters

The first version of Unicode was a 16-bit, fixed-width encoding that used two bytes to encode each character. This allowed 65,536 characters to be represented. However, more characters need to be supported, especially additional CJK ideographs that are important for the Chinese, Japanese, and Korean markets.

Unicode 3.1 defines supplementary characters to meet this need. It uses two 16-bit code points (also known as supplementary characters) to represent a single character. This enables an additional 1,048,576 characters to be defined. The Unicode 3.1 standard added the first group of 44,944 supplementary characters.

Adding supplementary characters increases the complexity of Unicode, but it is less complex than managing several different encodings in the same configuration.

Unicode Encodings

Unicode 3.1 encodes characters in different ways: UTF-8, UCS-2, and UTF-16. Conversion between different Unicode encodings is a simple bit-wise operation that is defined in the Unicode standard.

This section contains the following topics:

- UTF-8 Encoding
- UCS-2 Encoding
- UTF-16 Encoding
- Examples: UTF-16, UTF-8, and UCS-2 Encoding

UTF-8 Encoding

UTF-8 is the 8-bit encoding of Unicode. It is a variable-width encoding and a **strict superset** of ASCII. This means that each and every character in the ASCII character set is available in UTF-8 with the same code point values. One Unicode character can be 1 byte, 2 bytes, 3 bytes, or 4 bytes in UTF-8 encoding. Characters from the European scripts are represented in either 1 or 2 bytes. Characters from most Asian scripts are represented in 3 bytes. Supplementary characters are represented in 4 bytes.

UTF-8 is the Unicode encoding supported on UNIX platforms and used for HTML and most Internet browsers. Other environments such as Windows and Java use UCS-2 encoding.

The benefits of UTF-8 are as follows:

- Compact storage requirement for European scripts because it is a strict superset of ASCII
- Ease of migration between ASCII-based characters sets and UTF-8

See Also:

- "Supplementary Characters" on page 5-3
- Table B–2, "Unicode Character Code Ranges for UTF-8 Character Codes" on page B-2

UCS-2 Encoding

UCS-2 is a fixed-width, 16-bit encoding. Each character is 2 bytes. UCS-2 is the Unicode encoding used by Java and Microsoft Windows NT 4.0. UCS-2 supports characters defined for Unicode 3.0, so there is no support for supplementary characters.

The benefits of UCS-2 over UTF-8 are as follows:

- More compact storage for Asian scripts because all characters are two bytes
- Faster string processing because characters are fixed-width
- Better compatibility with Java and Microsoft clients

See Also: "Supplementary Characters" on page 5-3

UTF-16 Encoding

UTF-16 encoding is the 16-bit encoding of Unicode. UTF-16 is an extension of UCS-2 because it supports the supplementary characters that are defined in Unicode 3.1 by using two UCS-2 code points for each supplementary character. UTF-16 is a strict superset of UCS-2.

One character can be either 2 bytes or 4 bytes in UTF-16. Characters from European and most Asian scripts are represented in 2 bytes. Supplementary characters are represented in 4 bytes. UTF-16 is the main Unicode encoding used by Microsoft Windows 2000.

The benefits of UTF-16 over UTF-8 are as follows:

- More compact storage for Asian scripts because most of the commonly used Asian characters are represented in two bytes.
- Better compatibility with Java and Microsoft clients

See Also:

- "Supplementary Characters" on page 5-3
- Table B–1, "Unicode Character Code Ranges for UTF-16 Character Codes" on page B-2

Examples: UTF-16, UTF-8, and UCS-2 Encoding

Figure 5–1 shows some characters and their character codes in UTF-16, UTF-8, and UCS-2 encoding. The last character is a treble clef (a music symbol), a supplementary character that has been added to the Unicode 3.1 standard.

Figure 5–1 UTF-16, UTF-8, and UCS-2 Encoding Examples

Character	UTF-16	UTF-8	UCS-2
A	0041	41	0041
c	0063	63	0063
Ö	00F6	C3 B6	00F6
亜	4E9C	E4 BA 9C	4E9C
𝄞	D834 DD1E	F0 9D 84 9E	N/A

Oracle's Support for Unicode

Oracle Corporation started supporting Unicode as a database character set in Oracle7. Table 5–1 summarizes the Unicode character sets supported by the Oracle database server.

Table 5–1 Unicode Character Sets Supported by the Oracle Database Server

Character Set	Supported in RDBMS Release	Unicode Encoding	Unicode Version	Database Character Set	National Character Set
AL24UTFFSS	7.2 - 8*i*	UTF-8	1.1	Yes	No
UTF8	8.0 - 9*i*	UTF-8	For Oracle8 release 8.0 through Oracle8*i* release 8.1.6: 2.1 For Oracle8*i* release 8.1.7 and later: 3.0	Yes	Yes (Oracle9*i* only)
UTFE	8.0 - 9*i*	UTF-8	For Oracle8*i* releases 8.0 through 8.1.6: 2.1 For Oracle8*i* release 8.1.7 and later: 3.0	Yes	No
AL32UTF8	9*i*	UTF-8	Oracle9*i*, Release 1: 3.0 Oracle9*i*, Release 2: 3.1	Yes	No
AL16UTF16	9*i*	UTF-16	Oracle9*i*, Release 1: 3.0 Oracle9*i*, Release 2: 3.1	No	Yes

Implementing a Unicode Solution in the Database

You can store Unicode characters in an Oracle9*i* database in two ways.

You can create a Unicode database that enables you to store UTF-8 encoded characters as SQL `CHAR` datatypes (`CHAR`, `VARCHAR2`, `CLOB`, and `LONG`).

If you prefer to implement Unicode support incrementally or if you need to support multilingual data only in certain columns, then you can store Unicode data in either the UTF-16 or UTF-8 encoding form in SQL `NCHAR` datatypes (`NCHAR`, `NVARCHAR2`, and `NCLOB`). The SQL `NCHAR` datatypes are called Unicode datatypes because they are used only for storing Unicode data.

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

The following sections explain how to use the two Unicode solutions and how to choose between them:

- Enabling Multilingual Support with Unicode Databases
- Enabling Multilingual Support with Unicode Datatypes
- How to Choose Between a Unicode Database and a Unicode Datatype Solution
- Comparing Unicode Character Sets for Database and Datatype Solutions

Enabling Multilingual Support with Unicode Databases

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

- AL32UTF8

  The AL32UTF8 character set supports the latest version of the Unicode standard. It encodes characters in one, two, or three bytes. Supplementary characters require four bytes. It is for ASCII-based platforms.

- UTF8

  The UTF8 character set encodes characters in one, two, or three bytes. It is for ASCII-based platforms.

  The UTF8 character set has supported Unicode 3.0 since Oracle8*i* release 8.1.7 and will continue to support Unicode 3.0 in future releases of the Oracle database server. Although specific supplementary characters were not assigned code points in Unicode until version 3.1, the code point range was allocated for supplementary characters in Unicode 3.0. If supplementary characters are inserted into a UTF8 database, then it does not corrupt the data in the database. The supplementary characters are treated as two separate, user-defined characters that occupy 6 bytes. Oracle Corporation recommends that you switch to AL32UTF8 for full support of supplementary characters in the database character set.

- UTFE

  The UTFE character set is for EBCDIC platforms. It has the same properties as UTF8 on ASCII platforms.

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

To create a database with the AL32UTF8 character set, use the `CREATE DATABASE` statement and include the `CHARACTER SET AL32UTF8` clause. For example:

```
CREATE DATABASE sample
    CONTROLFILE REUSE
    LOGFILE
        GROUP 1 ('diskx:log1.log', 'disky:log1.log') SIZE 50K,
        GROUP 2 ('diskx:log2.log', 'disky:log2.log') SIZE 50K
    MAXLOGFILES 5
    MAXLOGHISTORY 100
    MAXDATAFILES 10
    MAXINSTANCES 2
    ARCHIVELOG
    CHARACTER SET AL32UTF8
    NATIONAL CHARACTER SET AL16UTF16
    DATAFILE
        'disk1:df1.dbf' AUTOEXTEND ON,
        'disk2:df2.dbf' AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
    DEFAULT TEMPORARY TABLESPACE temp_ts
    UNDO TABLESPACE undo_ts
    SET TIME_ZONE = '+02:00';
```

Note: Specify the database character set when you create the database.

Enabling Multilingual Support with Unicode Datatypes

An alternative to storing Unicode data in the database is to use the SQL `NCHAR` datatypes (`NCHAR`, `NVARCHAR`, `NCLOB`). You can store Unicode characters into columns of these datatypes regardless of how the database character set has been defined. The `NCHAR` datatype has been redefined in Oracle9*i* to be a Unicode datatype exclusively. In other words, it stores data encoded as Unicode.

In releases before Oracle9*i*, the `NCHAR` datatype supported fixed-width Asian character sets that were designed to provide higher performance. Examples of fixed-width character sets are JA16SJISFIXED and ZHT32EUCFIXED. No Unicode character set was supported as the national character set before Oracle9*i*.

You can create a table using the NVARCHAR2 and NCHAR datatypes. The column length specified for the NCHAR and NVARCHAR2 columns is always the number of characters instead of the number of bytes:

```
CREATE TABLE product_information
    ( product_id          NUMBER(6)
    , product_name        NVARCHAR2(100)
    , product_description VARCHAR2(1000));
```

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

- AL16UTF16

  This is the default character set for SQL NCHAR datatypes. The character set encodes Unicode data in the UTF-16 encoding. It supports supplementary characters, which are stored as four bytes.

- UTF8

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

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

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

```
CREATE DATABASE sample
    CONTROLFILE REUSE
    LOGFILE
        GROUP 1 ('diskx:log1.log', 'disky:log1.log') SIZE 50K,
        GROUP 2 ('diskx:log2.log', 'disky:log2.log') SIZE 50K
    MAXLOGFILES 5
    MAXLOGHISTORY 100
    MAXDATAFILES 10
    MAXINSTANCES 2
    ARCHIVELOG
    CHARACTER SET WE8ISO8859P1
    NATIONAL CHARACTER SET AL16UTF16
    DATAFILE
        'disk1:df1.dbf' AUTOEXTEND ON,
        'disk2:df2.dbf' AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
    DEFAULT TEMPORARY TABLESPACE temp_ts
    UNDO TABLESPACE undo_ts
    SET TIME_ZONE = '+02:00';
```

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

To choose the right Unicode solution for your database, consider the following questions:

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

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

- When Should You Use a Unicode Database?
- When Should You Use Unicode Datatypes?

When Should You Use a Unicode Database?

Use a Unicode database in the situations described in Table 5–2.

Table 5–2 Using a Unicode Database

Situation	Explanation
You need easy code migration for Java or PL/SQL.	If your existing application is mainly written in Java and PL/SQL and your main concern is to minimize the code changes required to support multiple languages, then you may want to use a Unicode database solution. If the datatypes used to stored data remain as SQL CHAR datatypes, then the Java and PL/SQL code that accesses these columns does not need to change.

Table 5–2 Using a Unicode Database (Cont.)

Situation	Explanation
You have evenly distributed multilingual data.	If the multilingual data will be evenly distributed in existing schema tables and you are not sure which ones will contain multilingual data, then you should use a Unicode database because it does not require you to identify the kind of data that is stored in each column.
Your SQL statements and PL/SQL code contain Unicode data.	You must use a Unicode database. SQL statements and PL/SQL code are converted into the database character set before being processed. If the SQL statements and PL/SQL code contain characters that cannot be converted to the database character set, then those characters will be lost. A common place to use Unicode data in a SQL statement is in a string literal.
You want to store multilingual documents as `BLOB`s and use Oracle Text for content searching.	You must use a Unicode database. The `BLOB` data is converted to the database character set before being indexed by Oracle Text. If your database character set is not UTF8, then data will be lost when the documents contain characters that cannot be converted to the database character set.

When Should You Use Unicode Datatypes?

Use Unicode datatypes in the situations described in Table 5–3.

Table 5–3 Using Unicode Datatypes

Situation	Explanation
You want to add multilingual support incrementally.	If you want to add Unicode support to the existing database without migrating the character set, then consider using Unicode datatypes to store Unicode data. You can add columns of the SQL `NCHAR` datatypes to existing tables or new tables to support multiple languages incrementally.
You want to build a packaged application.	If you are building a packaged application that will be sold to customers, then you may want to build the application using SQL `NCHAR` datatypes. The SQL `NCHAR` datatype is a reliable Unicode datatype in which the data is always stored in Unicode, and the length of the data is always specified in UTF-16 code units. As a result, you need to test the application only once. The application will run on customer databases with any database character set.

Table 5–3 Using Unicode Datatypes (Cont.)

Situation	Explanation
You want better performance with single-byte database character sets.	If performance is your main concern, then consider using a single-byte database character set and storing Unicode data in the SQL `NCHAR` datatypes. Databases that use a multibyte database character set such as UTF8 have a performance overhead.
You require UTF-16 support in Windows clients.	If your applications are written in Visual C/C++ or Visual Basic running on Windows, then you may want to use the SQL `NCHAR` datatypes. You can store UTF-16 data in SQL `NCHAR` datatypes in the same way that you store it in the `wchar_t` buffer in Visual C/C++ and `string` buffer in Visual Basic. You can avoid buffer overflow in client applications because the length of the `wchar_t` and `string` datatypes match the length of the SQL `NCHAR` datatypes in the database.

Note: You can use a Unicode database with Unicode datatypes.

Comparing Unicode Character Sets for Database and Datatype Solutions

Oracle9*i* provides two solutions to store Unicode characters in the database: a Unicode database solution and a Unicode datatype solution. After you select the Unicode database solution, the Unicode datatype solution or a combination of both, determine the character set to be used in the Unicode database or the Unicode datatype.

Table 5–4 contains advantages and disadvantages of different character sets for a Unicode database solution. The Oracle character sets that can be Unicode database character sets are AL32UTF8, UTF8, and UTFE.

Table 5–4 Character Set Advantages and Disadvantages for a Unicode Database Solution

Database Character Set	Advantages	Disadvantages
AL32UTF8	■ Supplementary characters are stored in 4 bytes, so there is no data conversion when supplementary characters are retrieved and inserted. ■ The storage for supplementary characters requires less disk space in AL32UTF8 than in UTF8.	■ You cannot specify the length of SQL CHAR types in number of characters (Unicode code points) for supplementary characters. Supplementary characters are treated as one code point rather than the standard two code points. ■ The binary order for SQL CHAR columns is different from the binary order of SQL NCHAR columns when the data consists of supplementary characters. As a result, CHAR columns and NCHAR columns do not always have the same sort for identical strings.
UTF8	■ You can specify the length of SQL CHAR types in number of characters. ■ The binary order of the SQL CHAR columns is always the same as the binary order of the SQL NCHAR columns when the data consists of the same supplementary characters. As a result, CHAR columns and NCHAR columns have the same sort for identical strings.	■ Supplementary characters are stored as 6 bytes instead of the 4 bytes defined by Unicode 3.1. As a result, Oracle has to convert data for supplementary characters.
UTFE	■ This is the only Unicode character set for the EBCDIC platform. ■ You can specify the length of SQL CHAR types in number of characters. ■ The binary order of the SQL CHAR columns is always the same as the binary order of the SQL NCHAR columns when the data consists of the same supplementary characters. As a result, CHAR columns and NCHAR columns have the same sort for identical strings.	■ Supplementary character are stored as 6 bytes instead of the 4 bytes defined by the Unicode standard. As a result, Oracle has to convert data for those supplementary characters. ■ UTFE is not a standard encoding in the Unicode standard. As a result, clients requiring standard UTF-8 encoding must convert data from UTFE to the standard encoding when data is retrieved and inserted.

Table 5–5 contains advantages and disadvantages of different character sets for a Unicode datatype solution. The Oracle character sets that can be national character sets are AL16UTF16 and UTF8.

Table 5–5 Character Set Advantages and Disadvantages for a Unicode Datatype Solution

National Character Set	Advantages	Disadvantages
AL16UTF16	▪ Asian data in AL16UTF16 is usually more compact than in UTF8. As a result, you save disk space and have less disk I/O when most of the multilingual data stored in the database is Asian data. ▪ It is usually faster to process strings encoded in the AL16UTF16 character set than strings encoded in UTF8 because Oracle9*i* processes most characters in an AL16UTF16 encoded string as fixed-width characters. ▪ The maximum length limits for the `NCHAR` and `NVARCHAR2` columns are 1000 and 2000 characters, respectively. Because the data is fixed-width, the lengths are guaranteed.	▪ European ASCII data requires more disk space to store in AL16UTF16 than in UTF8. If most of your data is European data, it uses more disk space than if it were UTF8 data. ▪ The maximum lengths for `NCHAR` and `NVARCHAR2` are 1000 and 2000 characters, which is less than the lengths for `NCHAR` (2000) and `NVARCHAR2` (4000) in UTF8.

Table 5–5 Character Set Advantages and Disadvantages for a Unicode Datatype Solution (Cont.)

National Character Set	Advantages	Disadvantages
UTF8	■ European data in UTF8 is usually more compact than in AL16UTF16. As a result, you will save disk space and have better response time when most of the multilingual data stored in the database is European data. ■ The maximum lengths for the NCHAR and NVARCHAR2 columns are 2000 and 4000 characters respectively, which is more than those for NCHAR (1000) and NVARCHAR2 (2000) in AL16UTF16. Although the maximum lengths of the NCHAR and NVARCHAR2 columns are larger in UTF8, the actual storage size is still bound by the byte limits of 2000 and 4000 bytes, respectively. For example, you can store 4000 UTF8 characters in an NVARCHAR2 column if all the characters are single byte, but only 4000/3 characters if all the characters are three bytes.	■ Asian data requires more disk space to store in UTF8 than in AL16UTF16. If most of your data is Asian data, then disk space usage is not as efficient as it is when the character set is AL16UTF16. ■ Although you can specify larger length limits for NCHAR and NVARCHAR, you are not guaranteed to be able to insert the number of characters specified by these limits. This is because UTF8 allows variable-width characters. ■ It is usually slower to process strings encoded in UTF8 than strings encoded in AL16UTF16 because UTF8 encoded strings consist of variable-width characters.

Unicode Case Studies

This section describes typical scenarios for storing Unicode characters in an Oracle9*i* database:

- Example 5–3, "Unicode Solution with a Unicode Database"
- Example 5–4, "Unicode Solution with Unicode Datatypes"
- Example 5–5, "Unicode Solution with a Unicode Database and Unicode Datatypes"

Example 5–3 Unicode Solution with a Unicode Database

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

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

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

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

Example 5–4 Unicode Solution with Unicode Datatypes

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

- The existing database has a database character set of WE8ISO8859P1.
- All character data in the existing database is composed of Western European characters.
- The database is around 50 GB.

A typical solution is take the following actions:

- Use `NCHAR` and `NVARCHAR2` datatypes to store Unicode characters
- Keep WE8ISO8859P1 as the database character set
- Use AL16UTF16 as the national character set

The reasons for this solution are:

- Migrating the existing database to a Unicode database required data conversion because the database character set is WE8ISO8859P1 (a Latin-1 character set), which is not a subset of UTF8. As a result, there would be some overhead in converting the data to UTF8.
- The additional languages are supported in new applications only. They do not depend on the existing applications or schemas. It is simpler to use the Unicode datatype in the new schema and keep the existing schemas unchanged.
- Only customer name columns require Unicode support. Using a single `NCHAR` column meets the customer's requirements without migrating the entire database.
- Because the languages to be supported are mostly Asian languages, AL16UTF16 should be used as the national character set so that disk space is used more efficiently.
- The lengths of the SQL `NCHAR` datatypes are defined as number of characters. This is the same as the way they are treated when using `wchar_t` strings in Windows C/C++ programs. This reduces programming complexity.
- Existing applications using the existing schemas are unaffected.

Example 5–5 Unicode Solution with a Unicode Database and Unicode Datatypes

A Japanese company wants to develop a new Java application on Oracle9*i*. The company expects that the application will support as many languages as possible in the long run.

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

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

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

Designing Database Schemas to Support Multiple Languages

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

- Specifying Column Lengths for Multilingual Data
- Storing Data in Multiple Languages
- Storing Documents in Multiple Languages in LOBs

- Creating Indexes for Searching Multilingual Document Contents

Specifying Column Lengths for Multilingual Data

When you use NCHAR and NVARCHAR2 datatypes for storing multilingual data, the column size specified for a column is defined in number of characters. (The number of characters means the number of Unicode code units.) Table 5–6 shows the maximum size of the NCHAR and NVARCHAR2 datatypes for the AL16UTF16 and UTF8 national character sets.

Table 5–6 Maximum Datatype Size

National Character Set	Maximum Column Size of NCHAR Datatype	Maximum Column Size of NVARCHAR2 Datatype
AL16UTF16	1000 characters	2000 characters
UTF8	2000 bytes	4000 bytes

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

Storing Data in Multiple Languages

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

There are many ways for a database schema to relate data to a language. The following sections provide different approaches:

- Store Language Information with the Data

- Select Translated Data Using Fine-Grained Access Control

Store Language Information with the Data

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

> **See Also:** Appendix A, "Locale Data" for a list of `NLS_LANGUAGE` values and their abbreviations

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

```
ALTER TABLE scott.product_information add (language_id VARCHAR2(50)):

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

Select Translated Data Using Fine-Grained Access Control

Fine-grained access control enables you to limit the degree to which a user can view information in a table or view. Typically, this is done by appending a `WHERE` clause. when you add a `WHERE` clause as a fine-grained access policy to a table or view, Oracle9*i* automatically appends the `WHERE` clause to any SQL statements on the table at run time so that only those rows satisfying the `WHERE` clause can be accessed.

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

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

Specify this `WHERE` clause as a fine-grained access policy for `product_information` as follows:

```
create function func1 ( sch varchar2 , obj varchar2 )
return varchar2(100);
begin
```

```
return 'language_id = sys_context(''userenv'', ''LANG'')';
end
/

DBMS_RLS.ADD_POLICY ('scott', 'product_information', 'lang_policy', 'scott',
'func1', 'select');
```

Then any SELECT statement on the product_information table automatically appends the WHERE clause.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information about fine-grained access control

Storing Documents in Multiple Languages in LOBs

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

Data in CLOB columns is stored as UCS-2 internally when the database character set is multibyte, such as UTF8 or AL32UTF8. Document contents are converted to UTF-16 when they are inserted into a CLOB column. This means that the storage space required for an English document doubles when the data is converted. Storage for an Asian language document in a CLOB column requires less storage space than the same document in a LONG column using UTF8, typically around 30% less, depending on the contents of the document.

Documents in NCLOB are also stored as UTF-16 regardless of the database character set or national character set. The storage space requirement is the same as for CLOBs. Document contents are converted to UTF-16 when they are inserted into a NCLOB column. If you want to store multilingual documents in a non-Unicode database, then choose NCLOB. However, content search on NCLOB is not yet supported.

Documents in BLOB format are stored as they are. No data conversion occurs during insertion and retrieval. However, SQL string manipulation functions (such as LENGTH or SUBSTR) and collation functions (such as NLS_SORT and ORDER BY) cannot be applied to the BLOB datatype.

Table 5–7 lists the advantages and disadvantages of the CLOB, NCLOB, and BLOB datatypes when storing documents:

Table 5–7 Comparison of LOB Datatypes for Document Storage

Datatypes	Advantages	Disadvantages
`CLOB`	▪ Content search support ▪ String manipulation support	▪ Depends on database character set ▪ Data conversion is necessary for insertion ▪ Cannot store binary documents
`NCLOB`	▪ Independent of database character set ▪ String manipulation support	▪ No content search support ▪ Data conversion is necessary for insertion ▪ Cannot store binary documents
`BLOB`	▪ Independent of database character set ▪ Content search support ▪ No data conversion, data stored as is ▪ Can store binary documents such as Microsoft Word or Microsoft Excel	▪ No string manipulation support

Creating Indexes for Searching Multilingual Document Contents

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

Create a multilexer to search multilingual documents. The multilexer chooses a language-specific lexer for each row, based on a language column. This section describe the high level steps to create indexes for documents in multiple languages. It contains the following topics:

- Creating Multilexers
- Creating Indexes for Documents Stored as CLOBs
- Creating Indexes for Documents Stored as BLOBs

See Also: *Oracle Text Reference*

Creating Multilexers

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

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

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

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

Now add the language-specific lexers to the multilexer preference using the `add_sub_lexer` call:

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

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

Creating Indexes for Documents Stored as CLOBs

The multilexer decides which lexer to use for each row based on a language column in the table. This is a character column that stores the language of the document in a text column. Use the Oracle language name to identify the language of a document in this column. For example, if you use `CLOB`s to store your documents, then add the language column to the table where the documents are stored:

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

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

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

Creating Indexes for Documents Stored as BLOBs

In addition to the language column, the character set and format columns must be added in the table where the documents are stored. The character set column stores the character set of the documents using the Oracle character set names. The format column specifies whether a document is a text or binary document. For example, the `CREATE TABLE` statement can specify columns called `characterset` and `format`:

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

You can put word-processing or spreadsheet documents into the table and specify `binary` in the `format` column. For documents in HTML, XML and text format, you can put them into the table and specify `text` in the `format` column.

Because there is a column in which to specify the character set, you can store text documents in different character sets.

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

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

You can use the `charset_filter` if all documents are in text format. The `charset_filter` converts data from the character set specified in the `charset` column to the database character set.

6
Programming with Unicode

This chapter describes how to use Oracle's database access products with Unicode. It contains the following topics:

Overview of Programming with Unicode

Oracle9*i* offers several database access products for inserting and retrieving Unicode data. Oracle offers database access products for commonly used programming environments such as Java and C/C++. Data is transparently converted between the database and client programs, which ensures that client programs are independent of the database character set and national character set. In addition, client programs are sometimes even independent of the character datatype, such as NCHAR or CHAR, used in the database.

To avoid overloading the database server with data conversion operations, Oracle9*i* always tries to move them to the client side database access products. In a few cases, data must be converted in the database, which affects performance. This chapter discusses details of the data conversion paths.

Database Access Product Stack and Unicode

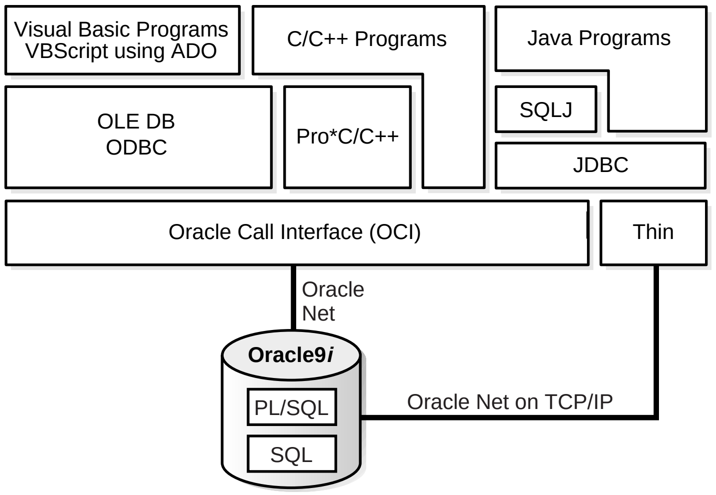
Oracle Corporation offers a comprehensive set of database access products that allow programs from different development environments to access Unicode data stored in the database. These products are listed in Table 6–1.

Table 6–1 Oracle Database Access Products

Programming Environment	Oracle Database Access Products
C/C++	Oracle Call Interface (OCI) Oracle Pro*C/C++ Oracle ODBC Driver Oracle OLE DB Driver
Visual Basic	Oracle ODBC Driver Oracle OLE DB Driver
Java	Oracle JDBC OCI or thin driver Oracle SQLJ
PL/SQL	Oracle PL/SQL and SQL

Figure 6–1 shows how the database access products can access the database.

Figure 6–1 Oracle Database Access Products



The Oracle Call Interface (OCI) is the lowest level API that the rest of the client-side database access products use. It provides a flexible way for C/C++ programs to access Unicode data stored in SQL `CHAR` and `NCHAR` datatypes. Using OCI, you can programmatically specify the character set (UTF-8, UTF-16, and others) for the data to be inserted or retrieved. It accesses the database through Oracle Net.

Oracle Pro*C/C++ enables you to embed SQL and PL/SQL in your programs. It uses OCI's Unicode capabilities to provide UTF-16 and UTF-8 data access for SQL `CHAR` and `NCHAR` datatypes.

The Oracle ODBC driver enables C/C++, Visual Basic, and VBScript programs running on Windows platforms to access Unicode data stored in SQL `CHAR` and `NCHAR` datatypes of the database. It provides UTF-16 data access by implementing the `SQLWCHAR` interface specified in the ODBC standard specification.

The Oracle OLE DB driver enables C/C++, Visual Basic, and VBScript programs running on Windows platforms to access Unicode data stored in SQL `CHAR` and `NCHAR` datatypes. It provides UTF-16 data access through wide string OLE DB datatypes.

Oracle JDBC drivers are the primary Java programmatic interface for accessing an Oracle9*i* database. Oracle provides two client-side JDBC drivers:

- The JDBC OCI driver that is used by Java applications and requires the OCI library

- The JDBC thin driver, which is a pure Java driver that is primarily used by Java applets and supports the Oracle Net protocol over TCP/IP

Both drivers support Unicode data access to SQL CHAR and NCHAR datatypes in the database.

Oracle SQLJ acts like a preprocessor that translates embedded SQL in a Java program into a Java source file with JDBC calls. It offers you a higher level programmatic interface to access databases. Like JDBC, SQLJ provides Unicode data access to SQL CHAR and NCHAR datatypes in the database.

The PL/SQL and SQL engines process PL/SQL programs and SQL statements on behalf of client-side programs such as OCI and server-side PL/SQL stored procedures. They allow PL/SQL programs to declare NCHAR and NVARCHAR2 variables and access SQL NCHAR datatypes in the database.

The following sections describe how each of the database access products supports Unicode data access to an Oracle9*i* database and offer examples for using those products:

- SQL and PL/SQL Programming with Unicode
- OCI Programming with Unicode
- Pro*C/C++ Programming with Unicode
- JDBC and SQLJ Programming with Unicode
- ODBC and OLE DB Programming with Unicode

SQL and PL/SQL Programming with Unicode

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

This section contains the following topics:

- SQL NCHAR Datatypes
- Implicit Datatype Conversion Between NCHAR and Other Datatypes
- Exception Handling for Data Loss During Datatype Conversion
- Rules for Implicit Datatype Conversion

- SQL Functions for Unicode Datatypes
- Other SQL Functions
- Unicode String Literals
- Using the UTL_FILE Package with NCHAR Data

See Also:

- *Oracle9i SQL Reference*
- *PL/SQL User's Guide and Reference*

SQL NCHAR Datatypes

There are three SQL NCHAR datatypes:

- The NCHAR Datatype
- The NVARCHAR2 Datatype
- The NCLOB Datatype

The NCHAR Datatype

When you define a table column or a PL/SQL variable as the NCHAR datatype, the length is always specified as the number of characters. For example, the statement

```
CREATE TABLE table1 (column1 NCHAR(30));
```

creates a column with a maximum length of 30 characters. The maximum number of bytes for the column is determined as follows:

```
maximum number of bytes = (maximum number of characters) x (maximum number of
bytes per character)
```

For example, if the national character set is UTF8, the maximum byte length is 30 characters times 3 bytes per character, or 90 bytes.

The national character set, which is used for all NCHAR datatypes, is defined when the database is created. In Oracle9*i*, the national character set can be either UTF8 or AL16UTF16. The default is AL16UTF16.

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

Note: UTF8 may affect performance because it is a variable-width character set. Excessive blank padding of NCHAR fields decreases performance. Consider using the NVARCHAR datatype or changing to the AL16UTF16 character set for the NCHAR datatype.

The NVARCHAR2 Datatype

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

The maximum column size allowed is 4000 characters when the national character set is UTF8 and 2000 when it is AL16UTF16. The maximum length of an NVARCHAR2 column in bytes is 4000. Both the byte limit and the character limit must be met, so the maximum number of characters that is actually allowed in an NVARCHAR2 column is the number of characters that can be written in 4000 bytes.

In PL/SQL, the maximum length for an NVARCHAR2 variable is 32767 bytes. You can define NVARCHAR2 variables up to 32767 characters, but the actual data cannot exceed 32767 bytes.

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

```
CREATE TABLE table2 (column2 NVARCHAR2(2000));
```

The NCLOB Datatype

NCLOB is a character large object containing multibyte characters, with a maximum size of 4 gigabytes. Unlike BLOBs, NCLOBs have full transactional support so that changes made through SQL, the DBMS_LOB package, or OCI participate fully in transactions.Manipulations of NCLOB value can be committed and rolled back.

Note, however, that you cannot save an NCLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

NCLOB values are stored in the database using the fixed-width AL16UTF16 character set, regardless of the national character set. Oracle translates the stored Unicode value to the character set requested on the client or on the server, which can be fixed-width or variable-width. When you insert data into an NCLOB column using a variable-width character set, Oracle converts the data into AL16UTF16 before storing it in the database.

> **See Also:** *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for more information about NCLOBs

Implicit Datatype Conversion Between NCHAR and Other Datatypes

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

Keep in mind these points about implicit conversions:

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

> **See Also:** *Oracle9i SQL Reference*

Exception Handling for Data Loss During Datatype Conversion

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

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

Rules for Implicit Datatype Conversion

In some cases, conversion between datatypes is possible in only one direction. In other cases, conversion in both directions is possible. Oracle defines a set of rules for conversion between datatypes. Table 6–2 contains the rules for conversion between datatypes.

Table 6–2 Rules for Conversion Between Datatypes

Statement	Rule
INSERT/UPDATE statement	Values are converted to the datatype of the target database column.
SELECT INTO statement	Data from the database is converted to the datatype of the target variable.
Variable assignments	Values on the right of the equal sign are converted to the datatype of the target variable on the left of the equal sign.
Parameters in SQL and PL/SQL functions	CHAR, VARCHAR2, NCHAR, and NVARCHAR2 are loaded the same way. An argument with a CHAR, VARCHAR2, NCHAR or NVARCHAR2 datatype is compared to a formal parameter of any of the CHAR, VARCHAR2, NCHAR or NVARCHAR2 datatypes. If the argument and formal parameter datatypes do not match exactly, then implicit conversions are introduced when data is copied into the parameter on function entry and copied out to the argument on function exit.
Concatenation || operation or CONCAT function	If one operand is a SQL CHAR or NCHAR datatype and the other operand is a NUMBER or other non-character datatype, then the other datatype is converted to VARCHAR2 or NVARCHAR2. For concatenation between character datatypes, see "SQL NCHAR datatypes and SQL CHAR datatypes" on page 6-9.
SQL CHAR or NCHAR datatypes and NUMBER datatype	Character value is converted to NUMBER datatype
SQL CHAR or NCHAR datatypes and DATE datatype	Character value is converted to DATE datatype

Table 6–2 Rules for Conversion Between Datatypes (Cont.)

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

SQL Functions for Unicode Datatypes

SQL NCHAR datatypes can be converted to and from SQL CHAR datatypes and other datatypes using explicit conversion functions. The examples in this section use the table created by the following statement:

```
CREATE TABLE customers
  (id NUMBER, name NVARCHAR2(50), address NVARCHAR2(200), birthdate DATE);
```

Example 6–1 Populating the Customer Table Using the TO_NCHAR Function

The `TO_NCHAR` function converts the data at run time, while the `N` function converts the data at compilation time.

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

Example 6–2 Selecting from the Customer Table Using the TO_CHAR Function

The following statement converts the values of `name` from characters in the national character set to characters in the database character set before selecting them according to the `LIKE` clause:

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

You should see the following output:

```
NAME
--------------------------------------
John Smith
```

Example 6–3 Selecting from the Customer Table Using the TO_DATE Function

Using the `N` function shows that either `NCHAR` or `CHAR` data can be passed as parameters for the `TO_DATE` function. The datatypes can mixed because they are converted at run time.

```
DECLARE
ndatestring NVARCHAR2(20) := N'12-SEP-1975';
BEGIN
SELECT name into ndstr FROM customers
WHERE (birthdate)> TO_DATE(ndatestring, 'DD-MON-YYYY', N'NLS_DATE_LANGUAGE =
AMERICAN');
END;
```

As demonstrated in Example 6–3, SQL `NCHAR` data can be passed to explicit conversion functions. SQL `CHAR` and `NCHAR` data can be mixed together when using multiple string parameters.

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

Other SQL Functions

Most SQL functions can take arguments of SQL NCHAR datatypes as well as mixed character datatypes. The return datatype is based on the type of the first argument. If a non-string datatype like NUMBER or DATE is passed to these functions, it will be converted to VARCHAR2. The following examples use the `customer` table created in "SQL Functions for Unicode Datatypes" on page 6-9.

Example 6–4 INSTR Function

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

Example 6–5 CONCAT Function

```
SELECT CONCAT(name,id) FROM customers;
```

`id` is converted to NVARCHAR2 and then concatenated with `name`.

Example 6–6 RPAD Function

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

The following output results:

```
RPAD(NAME,100,’’)
------------------------------------------
John Smith
```

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

See Also: *Oracle9i SQL Reference*

Unicode String Literals

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

- Put a prefix N in front of a single quote marked string literal. This explicitly indicates that the following string literal is an NCHAR string literal. For example, `N’12-SEP-1975’` is an NCHAR string literal.
- Mark a string literal with single quotations. Because Oracle supports implicit conversions to SQL NCHAR datatypes, a string literal is converted to a SQL NCHAR datatype wherever necessary.

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

- Use the `NCHR(`*`n`*`)` SQL function, which returns the character having the binary equivalent to *`n`* in the national character set, which is AL32UTF8 or AL16UTF16. The result of concatenating several `NCHR(`*`n`*`)` functions is `NVARCHAR2` data. In this way, you can bypass the client and server character set conversions and create an `NVARCHAR2` string directly. For example, `NCHR(32)` represents a blank character.

  Because `NCHR(`*`n`*`)` is associated with the national character set, portability of the resulting value is limited to applications that run in that national character set. If this is a concern, then use the `UNISTR` function to remove portability limitations.

- Use the `UNISTR`(*`string`*) SQL function. `UNISTR`(*`string`*) takes a string and converts it to Unicode. The result is in the national character set for the database. You can embed `escape \`*`bbbb`* inside the string. The escape represents the value of a UTF-16 code point with hex number 0x*`bbbb`*. For example, `UNISTR('G\0061ry')` represents `'Gary'`.

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

Using the UTL_FILE Package with NCHAR Data

The `UTL_FILE` package has been enhanced in Oracle9*i* to handle Unicode national character set data. The following functions and procedures have been added:

- `FOPEN_NCHAR`

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

- `GET_LINE_NCHAR`

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

- `PUT_NCHAR`

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

- `PUT_LINE_NCHAR`

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

- `PUTF_NCHAR`

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

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

OCI Programming with Unicode

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

- OCIEnvNlsCreate() Function for Unicode Programming
- OCI Unicode Code Conversion
- When the NLS_LANG Character Set is UTF8 or AL32UTF8 in OCI
- Binding and Defining SQL CHAR Datatypes in OCI
- Binding and Defining SQL NCHAR Datatypes in OCI
- Binding and Defining CLOB and NCLOB Unicode Data in OCI

See Also: Chapter 8, "OCI Programming in a Global Environment"

OCIEnvNlsCreate() Function for Unicode Programming

The `OCIEnvNlsCreate()` function is used to specify a SQL `CHAR` character set and a SQL `NCHAR` character set when the OCI environment is created. It is an enhanced version of the `OCIEnvCreate()` function and has extended arguments for two character set IDs. The OCI_UTF16ID UTF-16 character set ID replaces the Unicode mode introduced in Oracle9*i* release 1 (9.0.1). For example:

```
OCIEnv *envhp;
status = OCIEnvNlsCreate((OCIEnv **)&envhp,
(ub4)0,
(void *)0,
(void *(*) ()) 0,
(void *(*) ()) 0,
(void(*) ()) 0,
(size_t) 0,
(void **)0,
(ub2)OCI_UTF16ID, /* Metadata and SQL CHAR character set */
(ub2)OCI_UTF16ID /* SQL NCHAR character set */);
```

The Unicode mode, in which the OCI_UTF16 flag is used with the `OCIEnvCreate()` function, is deprecated.

When OCI_UTF16ID is specified for both SQL `CHAR` and SQL `NCHAR` character sets, all metadata and bound and defined data are encoded in UTF-16. Metadata includes SQL statements, user names, error messages, and column names. Thus, all inherited operations are independent of the `NLS_LANG` setting, and all metatext data parameters (`text*`) are assumed to be Unicode text datatypes (`utext*`) in UTF-16 encoding.

To prepare the SQL statement when the `OCIEnv()` function is initialized with the OCI_UTF16ID character set ID, call the `OCIStmtPrepare()` function with a `(utext*)` string. The following example runs on the Windows platform only. You may need to change `wchar_t` datatypes for other platforms.

```
const wchar_t sqlstr[] = L"SELECT * FROM ENAME=:ename";
...
OCIStmt* stmthp;
sts = OCIHandleAlloc(envh, (void **)&stmthp, OCI_HTYPE_STMT, 0,
NULL);
status = OCIStmtPrepare(stmthp, errhp,(const text*)sqlstr,
wcslen(sqlstr),
                        OCI_NTV_SYNTAX, OCI_DEFAULT);
```

To bind and define data, you do not have to set the `OCI_ATTR_CHARSET_ID` attribute because the `OCIEnv()` function has already been initialized with UTF-16 character set IDs. The bind variable names must be also UTF-16 strings.

```
/* Inserting Unicode data */
OCIBindByName(stmthp1, &bnd1p, errhp, (const text*)L":ename",
(sb4)wcslen(L":ename"),
              (void *) ename, sizeof(ename), SQLT_STR, (void
*)&insname_ind,
              (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *)0,
OCI_DEFAULT);
OCIAttrSet((void *) bnd1p, (ub4) OCI_HTYPE_BIND, (void *)
&ename_col_len,
           (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving Unicode data */
OCIDefineByPos (stmthp2, &dfn1p, errhp, (ub4)1, (void *)ename,
               (sb4)sizeof(ename), SQLT_STR, (void *)0, (ub2 *)0,
(ub2*)0,
            (ub4)OCI_DEFAULT);
```

The `OCIExecute()` function performs the operation.

See Also: "OCIEnvNlsCreate()" on page 8-2

OCI Unicode Code Conversion

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

Data Integrity

You can lose data during conversion if you call an OCI API inappropriately. If the server and client character sets are different, you can lose data when the destination character set is a smaller set than the source character set. You can avoid this potential problem if both character sets are Unicode character sets (for example, UTF8 and AL16UTF16).

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

OCI Performance Implications When Using Unicode

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

- When you bind or define SQL `CHAR` datatypes and set the `OCI_ATTR_CHARSET_FORM` attribute to `SQLCS_NCHAR`, data conversions take place from client character set to the national database character set, and from the national character set to the database character set. No data loss is expected, but two conversions happen, even though it requires only one.

- When you bind or define SQL `NCHAR` datatypes and do not set `OCI_ATTR_CHARSET_FORM`, data conversions take place from client character set to the database character set, and from the database character set to the national database character set. In the worst case, data loss can occur if the database character set is smaller than the client's.

To avoid performance problems, you should always set `OCI_ATTR_CHARSET_FORM` correctly, based on the datatype of the target columns. If you do not know the target datatype, you should set the `OCI_ATTR_CHARSET_FORM` attribute to `SQLCS_NCHAR` when binding and defining.

Table 6–3 contains information about OCI character set conversions.

Table 6–3 OCI Character Set Conversions

Datatypes for OCI Client Buffer	OCI_ATTR_CHARSET_FORM	Datatypes of the Target Column in the Database	Conversion Between	Comments
`utext`	`SQLCS_IMPLICIT`	`CHAR, VARCHAR2, CLOB`	UTF-16 and database character set in OCI	No unexpected data loss
`utext`	`SQLCS_NCHAR`	`NCHAR, NVARCHAR2, NCLOB`	UTF-16 and national character set in OCI	No unexpected data loss
`utext`	`SQLCS_NCHAR`	`CHAR, VARCHAR2, CLOB`	UTF-16 and national character set in OCI National character set and database character set in database server	No unexpected data loss, but may degrade performance because the conversion goes through the national character set
`utext`	`SQLCS_IMPLICIT`	`NCHAR, NVARCHAR2, NCLOB`	UTF-16 and database character set in OCI Database character set and national character set in database server	Data loss may occur if the database character set is not Unicode
`text`	`SQLCS_IMPLICIT`	`CHAR, VARCHAR2, CLOB`	`NLS_LANG` character set and database character set in OCI	No unexpected data loss

Table 6–3 OCI Character Set Conversions (Cont.)

Datatypes for OCI Client Buffer	OCI_ATTR_ CHARSET_ FORM	Datatypes of the Target Column in the Database	Conversion Between	Comments
`text`	`SQLCS_ NCHAR`	`NCHAR, NVARCHAR2,NCLOB`	`NLS_LANG` character set and national character set in OCI	No unexpected data loss
`text`	`SQLCS_ NCHAR`	`CHAR, VARCHAR2, CLOB`	`NLS_LANG` character set and national character set in OCI National character set and database character set in database server	No unexpected data loss, but may degrade performance because the conversion goes through the national character set
`text`	`SQLCS_ IMPLICIT`	`NCHAR, NVARCHAR2,NCLOB`	`NLS_LANG` character set and database character set in OCI Database character set and national character set in database server	Data loss may occur because the conversion goes through the database character set

OCI Unicode Data Expansion

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

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

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

```
ub2 csid = OCI_UTF16ID;
oratext *selstmt = "SELECT ename FROM emp";
counter = 1;
```

```
...
OCIStmtPrepare(stmthp, errhp, selstmt, (ub4)strlen((char*)selstmt),
               OCI_NTV_SYNTAX, OCI_DEFAULT);
OCIStmtExecute ( svchp, stmthp, errhp, (ub4)0, (ub4)0,
                 (CONST OCISnapshot*)0, (OCISnapshot*)0,
                 OCI_DESCRIBE_ONLY);
OCIParamGet(stmthp, OCI_HTYPE_STMT, errhp, &myparam, (ub4)counter);
OCIAttrGet((void*)myparam, (ub4)OCI_DTYPE_PARAM, (void*)&col_width,
           (ub4*)0, (ub4)OCI_ATTR_DATA_SIZE, errhp);
...
maxenamelen = (col_width + 1) * sizeof(utext);
cbuf = (utext*)malloc(maxenamelen);
...
OCIDefineByPos(stmthp, &dfnp, errhp, (ub4)1, (void *)cbuf,
                (sb4)maxenamelen, SQLT_STR, (void *)0, (ub2 *)0,
                (ub2*)0, (ub4)OCI_DEFAULT);
OCIAttrSet((void *) dfnp, (ub4) OCI_HTYPE_DEFINE, (void *) &csid,
           (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIStmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT);
...
```

When the NLS_LANG Character Set is UTF8 or AL32UTF8 in OCI

You can use UTF8 and AL32UTF8 by setting `NLS_LANG` for OCI client applications. If you do not need supplementary characters, then it does not matter whether you choose UTF8 or AL32UTF8. However, if your OCI applications might handle supplementary characters, then you need to make a decision. Because UTF8 can require up to three bytes for each character, one supplementary character is represented in two code points, totalling six bytes. In AL32UTF8, one supplementary character is represented in one code point, totalling four bytes.

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

Binding and Defining SQL CHAR Datatypes in OCI

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

- Call `OCIBind()` or `OCIDefine()` followed by `OCIAttrSet()` to specify UTF-16 Unicode character set encoding. For example:

```
...
ub2 csid = OCI_UTF16ID;
utext ename[100]; /* enough buffer for ENAME */
...
/* Inserting Unicode data */
OCIBindByName(stmthp1, &bnd1p, errhp, (oratext*)":ENAME",
             (sb4)strlen((char *)":ENAME"), (void *) ename, sizeof(ename),
             SQLT_STR, (void *)&insname_ind, (ub2 *) 0, (ub2 *) 0, (ub4) 0,
             (ub4 *)0, OCI_DEFAULT);
OCIAttrSet((void *) bnd1p, (ub4) OCI_HTYPE_BIND, (void *) &csid,
           (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAttrSet((void *) bnd1p, (ub4) OCI_HTYPE_BIND, (void *) &ename_col_len,
           (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving Unicode data */
OCIDefineByPos (stmthp2, &dfn1p, errhp, (ub4)1, (void *)ename,
                (sb4)sizeof(ename), SQLT_STR, (void *)0, (ub2 *)0,
                (ub2*)0, (ub4)OCI_DEFAULT);
OCIAttrSet((void *) dfn1p, (ub4) OCI_HTYPE_DEFINE, (void *) &csid,
           (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
...
```

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

- Call `OCIBind()` or `OCIDefine()` with the `NLS_LANG` character set specified as UTF8 or AL32UTF8.

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

```
...
oratext ename[100]; /* enough buffer size for ENAME */
...
/* Inserting Unicode data */
OCIBindByName(stmthp1, &bnd1p, errhp, (oratext*)":ENAME",
              (sb4)strlen((char *)":ENAME"), (void *) ename, sizeof(ename),
              SQLT_STR, (void *)&insname_ind, (ub2 *) 0, (ub2 *) 0,
              (ub4) 0, (ub4 *)0, OCI_DEFAULT);
OCIAttrSet((void *) bnd1p, (ub4) OCI_HTYPE_BIND, (void *) &ename_col_len,
           (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving Unicode data */
OCIDefineByPos (stmthp2, &dfn1p, errhp, (ub4)1, (void *)ename,
                (sb4)sizeof(ename), SQLT_STR, (void *)0, (ub2 *)0, (ub2*)0,
                (ub4)OCI_DEFAULT);
...
```

Binding and Defining SQL NCHAR Datatypes in OCI

Oracle Corporation recommends that you access SQL NCHAR datatypes using UTF-16 binding or defining when using OCI. Starting from Oracle9*i*, SQL NCHAR datatypes are Unicode datatypes with an encoding of either UTF8 or AL16UTF16. To access data in SQL NCHAR datatypes, set the OCI_ATTR_CHARSET_FORM attribute to SQLCS_NCHAR between binding or defining and execution so that it performs an appropriate data conversion without data loss. The length of data in SQL NCHAR datatypes is always in the number of Unicode code points.

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

```
...
ub2 csid = OCI_UTF16ID;
ub1 cform = SQLCS_NCHAR;
utext ename[100]; /* enough buffer for ENAME */
...
/* Inserting Unicode data */
OCIBindByName(stmthp1, &bnd1p, errhp, (oratext*)":ENAME",
              (sb4)strlen((char *)":ENAME"), (void *) ename,
              sizeof(ename), SQLT_STR, (void *)&insname_ind, (ub2 *) 0,
              (ub2 *) 0, (ub4) 0, (ub4 *)0, OCI_DEFAULT);
OCIAttrSet((void *) bnd1p, (ub4) OCI_HTYPE_BIND, (void *) &cform, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_FORM, errhp);
OCIAttrSet((void *) bnd1p, (ub4) OCI_HTYPE_BIND, (void *) &csid, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAttrSet((void *) bnd1p, (ub4) OCI_HTYPE_BIND, (void *) &ename_col_len,
           (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving Unicode data */
OCIDefineByPos (stmthp2, &dfn1p, errhp, (ub4)1, (void *)ename,
                (sb4)sizeof(ename), SQLT_STR, (void *)0, (ub2 *)0, (ub2*)0,
                (ub4)OCI_DEFAULT);
```

```
OCIAttrSet((void *) dfn1p, (ub4) OCI_HTYPE_DEFINE, (void *) &csid, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAttrSet((void *) dfn1p, (ub4) OCI_HTYPE_DEFINE, (void *) &cform, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_FORM, errhp);
...
```

Binding and Defining CLOB and NCLOB Unicode Data in OCI

In order to write (bind) and read (define) UTF-16 data for `CLOB` or `NCLOB` columns, the UTF-16 character set ID must be specified as `OCILobWrite()` and `OCILobRead()`. When you write UTF-16 data into a `CLOB` column, call `OCILobWrite()` as follows:

```
...
ub2 csid = OCI_UTF16ID;
err = OCILobWrite (ctx->svchp, ctx->errhp, lobp, &amtp, offset, (void *) buf,
                   (ub4) BUFSIZE, OCI_ONE_PIECE, (void *)0,
                   (sb4 (*)()) 0, (ub2) csid, (ub1) SQLCS_IMPLICIT);
```

The `amtp` parameter is the data length in number of Unicode code points. The `offset` parameter indicates the offset of data from the beginning of the data column. The `csid` parameter must be set for UTF-16 data.

To read UTF-16 data from `CLOB` columns, call `OCILobRead()` as follows:

```
...
ub2 csid = OCI_UTF16ID;
err = OCILobRead(ctx->svchp, ctx->errhp, lobp, &amtp, offset, (void *) buf,
                 (ub4)BUFSIZE , (void *) 0, (sb4 (*)()) 0, (ub2)csid,
                 (ub1) SQLCS_IMPLICIT);
```

The data length is always represented in the number of Unicode code points. Note one Unicodesupplementary character is counted as two code points, because the encoding is UTF-16. After binding or defining `LOB` column, you can measure the data length stored in the `LOB` column using `OCILobGetLength()`. The returning value is the data length in the number of code points if you bind or define as UTF-16.

```
err = OCILobGetLength(ctx->svchp, ctx->errhp, lobp, &lenp);
```

If you are using an `NCLOB`, you must set `OCI_ATTR_CHARSET_FORM` to `SQLCS_NCHAR`.

Pro*C/C++ Programming with Unicode

Pro*C/C++ provides the following ways to insert or retrieve Unicode data into or from the database:

- Using the VARCHAR Pro*C/C++ datatype or the native C/C++ `text` datatype, a program can access Unicode data stored in SQL CHAR datatypes of a UTF8 or AL32UTF8 database. Alternatively, a program could use the C/C++ native `text` type.
- Using the UVARCHAR Pro*C/C++ datatype or the native C/C++ `utext` datatype, a program can access Unicode data stored in NCHAR datatypes of a database.
- Using the NVARCHAR Pro*C/C++ datatype, a program can access Unicode data stored in NCHAR datatypes. The difference between UVARCHAR and NVARCHAR in a Pro*C/C++ program is that the data for the UVARCHAR datatype is stored in a `utext` buffer while the data for the NVARCHAR datatype is stored in a `text` datatype.

Pro*C/C++ does not use the Unicode OCI API for SQL text. As a result, embedded SQL text must be encoded in the character set specified in the NLS_LANG environment variable.

This section contains the following topics:

- Pro*C/C++ Data Conversion in Unicode
- Using the VARCHAR Datatype in Pro*C/C++
- Using the NVARCHAR Datatype in Pro*C/C++
- Using the UVARCHAR Datatype in Pro*C/C++

Pro*C/C++ Data Conversion in Unicode

Data conversion occurs in the OCI layer, but it is the Pro*C/C++ preprocessor that instructs OCI which conversion path should be taken based on the datatypes used in a Pro*C/C++ program. Table 6–4 illustrates the conversion paths:

Table 6–4 Pro*C/C++ Bind and Define Data Conversion

Pro*C/C++ Datatype	SQL Datatype	Conversion Path
VARCHAR or text	CHAR	NLS_LANG character set to and from the database character set happens in OCI
VARCHAR or text	NCHAR	NLS_LANG character set to and from database character set happens in OCI Database character set to and from national character set happens in database server
NVARCHAR	NCHAR	NLS_LANG character set to and from national character set happens in OCI
NVARCHAR	CHAR	NLS_LANG character set to and from national character set happens in OCI National character set to and from database character set in database server
UVARCHAR or utext	NCHAR	UTF-16 to and from the national character set happens in OCI
UVARCHAR or utext	CHAR	UTF-16 to and from national character set happens in OCI National character set to database character set happens in database server

Using the VARCHAR Datatype in Pro*C/C++

The Pro*C/C++ VARCHAR datatype is preprocessed to a struct with a length field and text buffer field. The following example uses the C/C++ text native datatype and the VARCHAR Pro*C/C++ datatypes to bind and define table columns.

```
#include <sqlca.h>
main()
{
   ...
   /* Change to STRING datatype:    */
   EXEC ORACLE OPTION (CHAR_MAP=STRING) ;
   text ename[20] ;                  /* unsigned short type */
   varchar address[50] ;             /* Pro*C/C++ uvarchar type */

   EXEC SQL SELECT ename, address INTO :ename, :address FROM emp;
   /* ename is NULL-terminated */
   printf(L"ENAME = %s, ADDRESS = %.*s\n", ename, address.len, address.arr);
   ...
}
```

When you use the VARCHAR datatype or native text datatype in a Pro*C/C++ program, the preprocessor assumes that the program intends to access columns of SQL CHAR datatypes instead of SQL NCHAR datatypes in the database. The preprocessor generates C/C++ code to reflect this fact by doing a bind or define using the SQLCS_IMPLICIT value for the OCI_ATTR_CHARSET_FORM attribute. As a result, if a bind or define variable is bound to a column of SQL NCHAR datatypes in the database, implicit conversion happens in the database server to convert the data from the database character set to the national database character set and vice versa. During the conversion, data loss occurs when the database character set is a smaller set than the national character set.

Using the NVARCHAR Datatype in Pro*C/C++

The Pro*C/C++ NVARCHAR datatype is similar to the Pro*C/C++ VARCHAR datatype. It should be used to access SQL NCHAR datatypes in the database. It tells Pro*C/C++ preprocessor to bind or define a text buffer to the column of SQL NCHAR datatypes. The preprocessor will specify the SQLCS_NCHAR value for the OCI_ATTR_CHARSET_FORM attribute of the bind or define variable. As a result, no implicit conversion occurs in the database.

If the NVARCHAR buffer is bound against columns of SQL CHAR datatypes, the data in the NVARCHAR buffer (encoded in the NLS_LANG character set) is converted to or from the national character set in OCI, and the data is then converted to the database character set in the database server. Data can be lost when the NLS_LANG character set is a larger set than the database character set.

Using the UVARCHAR Datatype in Pro*C/C++

The UVARCHAR datatype is preprocessed to a struct with a length field and utext buffer field. The following example code contains two host variables, ename and address. The ename host variable is declared as a utext buffer containing 20 Unicode characters. The address host variable is declared as a uvarchar buffer containing 50 Unicode characters, the len and arr fields are accessible as fields of a struct.

```
#include <sqlca.h>
#include <sqlucs2.h>

main()
{
   ...
   /* Change to STRING datatype:    */
   EXEC ORACLE OPTION (CHAR_MAP=STRING) ;
   utext ename[20] ;                 /* unsigned short type */
 uvarchar address[50] ;              /* Pro*C/C++ uvarchar type */

   EXEC SQL SELECT ename, address INTO :ename, :address FROM emp;
   /* ename is NULL-terminated */
wprintf(L"ENAME = %s, ADDRESS = %.*s\n", ename, address.len,
address.arr);
...
}
```

When you use the UVARCHAR datatype or native utext datatype in Pro*C/C++ programs, the preprocessor assumes that the program intends to access SQL NCHAR datatypes. The preprocessor generates C/C++ code by binding or defining using the SQLCS_NCHAR value for OCI_ATTR_CHARSET_FORM attribute. As a result, if a bind or define variable is bound to a column of a SQL NCHAR datatype, an implicit conversion of the data from the national character set occurs in the database server. However, there is no data lost in this scenario because the national character set is always a larger set than the database character set.

JDBC and SQLJ Programming with Unicode

Oracle provides three JDBC drivers for Java programs to access Unicode data in the database:

- The JDBC OCI driver
- The JDBC thin driver
- The JDBC KPRB driver

Java programs can insert or retrieve Unicode data to and from columns of SQL CHAR and NCHAR datatypes. Specifically, JDBC enables Java programs to bind or define Java strings to SQL CHAR and NCHAR datatypes. Because Java's string datatype is UTF-16 encoded, data retrieved from or inserted into the database must be converted from UTF-16 to the database character set or the national character set and vice versa. The SQLJ preprocessor enables Java programs to embed SQL statements to simplify database access code. It translates the embedded SQL statements of a Java program to the corresponding JDBC calls. Similar to JDBC, SQLJ enables programs to bind or define Java strings to a SQL CHAR or NCHAR column. JDBC and SQLJ also allow you to specify the PL/SQL and SQL statements in Java strings so that any non-ASCII schema object names can be referenced in Java programs.

This section contains the following topics:

- Binding and Defining Java Strings in Unicode
- Java Data Conversion in Unicode
- Java Data Conversion in Unicode

See Also: Chapter 9, "Java Programming in a Global Environment"

Binding and Defining Java Strings in Unicode

Oracle JDBC drivers allow you to access SQL CHAR datatypes in the database using Java string bind or define variables. The following code illustrates how to bind or define a Java string to a CHAR column:

```
int empno = 12345;
String ename = "Joe"
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO" +
    "emp (ename, empno) VALUES (?, ?)");
pstmt.setString(1, ename);
pstmt.setInt(2, empno);
pstmt.execute();                /* execute to insert into first row */
empno += 1;                     /* next employee number */
ename = "\uFF2A\uFF4F\uFF45";    /* Unicode characters in name */
pstmt.setString(1, ename);
pstmt.setInt(2, empno);
pstmt.execute();                /* execute to insert into second row */
```

For binding or defining Java string variables to SQL NCHAR datatypes, Oracle extends the JDBC specification to add the `PreparedStatement.setFormOfUse()` method through which you can explicitly specify the target column of a bind variable to be a SQL NCHAR datatype. The following code illustrates how to bind a Java string to an NCHAR column:

```
int empno = 12345;
String ename = "Joe"
oracle.jdbc.OraclePreparedStatement pstmt =
    (oracle.jdbc.OraclePreparedStatement)
    conn.prepareStatement("INSERT INTO emp (ename, empno) VALUES (?, ?)");
pstmt.setFormOfUse(1, oracle.jdbc.OraclePreparedStatement.FORM_NCHAR);
pstmt.setString(1, ename);
pstmt.setInt(2, empno);
pstmt.execute();                /* execute to insert into first row */
empno += 1;                     /* next employee number */
```

```
ename = "\uFF2A\uFF4F\uFF45";    /* Unicode characters in name */
pstmt.setString(1, ename);
pstmt.setInt(2, empno);
pstmt.execute();                 /* execute to insert into second row */
```

You can bind or define a Java string against an `NCHAR` column without explicitly specifying the form of use argument. This implies the following:

- If you do not specify the argument in the `setString()` method, JDBC assumes that the bind or define variable is for the SQL `CHAR` column. As a result, it tries to convert them to the database character set. When the data gets to the database, the database implicitly converts the data in the database character set to the national character set. During this conversion, data can be lost when the database character set is a subset of the national character set. Because the national character set is either UTF8 or AL16UTF16, data loss would happen if the database character set is not UTF8 or AL32UTF8.
- Because implicit conversion from SQL `CHAR` to SQL `NCHAR` datatypes happens in the database, database performance is degraded.

In addition, if you bind or define a Java string for a column of SQL `CHAR` datatypes but specify the form of use argument, performance of the database will be degraded. However, data should not be lost because the national character set is always a larger set than the database character set.

Java Data Conversion in Unicode

Because Java strings are always encoded in UTF-16, JDBC drivers transparently convert data from the database character set to UTF-16 or the national character set.The conversion paths taken are different for the three JDBC drivers:

- Data Conversion for the OCI Driver
- Data Conversion for the Thin Driver
- Data Conversion for the JDBC Driver

Data Conversion for the OCI Driver

For the OCI driver, the SQL statements are always converted to the database character set by the driver before it is sent to the database for processing. For Java `string` bind or define variables, Table 6–5 summarizes the conversion paths taken for different scenarios:

Table 6–5 OCI Driver Conversion Path

Form of Use	SQL Datatype	Conversion Path
`Const.CHAR (Default)`	`CHAR`	Java String to and from database character set happens in the JDBC driver
`Const.CHAR (Default)`	`NCHAR`	Java String to and from database character set happens in the JDBC driver. Data in the database character set to and from national character set happens in the database server
`Const.NCHAR`	`NCHAR`	Java String to and from national character set happens in the JDBC driver
`Const.NCHAR`	`CHAR`	Java String to and from national character set happens in the JDBC driver Data in national character set to and from database character set happens in the database server

Data Conversion for the Thin Driver

For the thin driver, SQL statements are always converted to either the database character set or to UTF-8 by the driver before they are sent to the database for processing. The thin driver also notifies the database that a SQL statement requires further conversion before being processed. The database, in turn, converts the SQL statement to the database character set. For Java `string` bind and define variables, the conversion paths shown in Table 6–6 are taken for the thin driver:

Table 6–6 Thin Driver Conversion Path

Form of Use	SQL Datatype	Database Character Set	Conversion Path
`Const.CHAR (Default)`	`CHAR`	US7ASCII or WE8ISO8859P1	Java String to and from the database character set happens in the thin driver
`Const.CHAR (Default)`	`NCHAR`	US7ASCII or WE8ISO8859P1	Java String to and from the database character set happens in the thin driver. Data in the database character set to and from the national character set happens in the database server
`Const.CHAR (Default)`	`CHAR`	non-ASCII and non-WE8ISO8859P1	Java String to and from UTF-8 happens in the thin driver. Data in UTF-8 to and from the database character set happens in the database server
`Const.CHAR (Default)`	`CHAR`	non-ASCII and non-WE8ISO8859P1	Java String to and from UTF-8 happens in the thin driver. Data in UTF-8 to and from national character set happens in the database server
`Const.NCHAR`	`CHAR`		Java String to and from the national character set happens in the thin driver. Data in the national character set to and from the database character set happens in the database server
`Const.NCHAR`	`NCHAR`		Java String to and from the national character set happens in the thin driver

Data Conversion for the JDBC Driver

The JDBC server-side internal driver runs in the server. All conversions are done in the database server. SQL statements specified as Java strings are converted to the database character set. Java `string` bind or define variables are converted to the database character sets if the form of use argument is not specified. Otherwise, they are converted to the national character set.

ODBC and OLE DB Programming with Unicode

You should use Oracle's ODBC and OLE DB drivers to access Oracle9*i* when using a Windows platform. This section describes how these drivers support Unicode. It includes the following topics:

Unicode-Enabled Drivers in ODBC and OLE DB

Oracle's ODBC and OLE DB drivers can handle Unicode data properly without data loss. For example, you can run a Unicode ODBC application containing Japanese data on English Windows if you install Japanese fonts and an input method editor for entering Japanese characters.

In Oracle9*i*, Oracle provides Windows platform-specific ODBC and OLE DB drivers only. For Unix platforms, contact your vendor.

OCI Dependency in Unicode

OCI Unicode binding and defining features are used by the ODBC and OLE DB drivers to handle Unicode data. OCI Unicode data binding and defining features are independent from `NLS_LANG`. This means Unicode data is handled properly, irrespective of the `NLS_LANG` setting on the platform.

See Also: "OCI Programming with Unicode" on page 6-13

ODBC and OLE DB Code Conversion in Unicode

In general, no redundant data conversion occurs unless you specify a different client datatype from that of the server. If you bind Unicode buffer `SQL_C_WCHAR` with a Unicode data column like `NCHAR`, for example, ODBC and OLE DB drivers bypass it between the application and OCI layer.

If you do not specify datatypes before fetching, but call `SQLGetData` with the client datatypes instead, then the conversions in Table 6–7 occur.

Table 6–7 ODBC Implicit Binding Code Conversions

Datatypes of ODBC Client Buffer	Datatypes of the Target Column in the Database	Fetch Conversions	Comments
`SQL_C_WCHAR`	`CHAR`, `VARCHAR2`, `CLOB`	[If the database character set is a subset of the `NLS_LANG` character set, then the conversions occur in the following order: ■ Database character set ■ `NLS_LANG` ■ UTF-16 in OCI ■ UTF-16 in ODBC	No unexpected data loss May degrade performance if database character set is a subset of the `NLS_LANG` character set
`SQL_C_CHAR`	`CHAR`, `VARCHAR2`, `CLOB`	If database character set is a subset of `NLS_LANG` character set: Database character set to `NLS_LANG` in OCI If database character set is NOT a subset of `NLS_LANG` character set: Database character set, UTF-16, to `NLS_LANG` character set in OCI and ODBC	No unexpected data loss May degrade performance if database character set is not a subset of `NLS_LANG` character set

You must specify the datatype for inserting and updating operations.

The datatype of the ODBC client buffer is given when you call `SQLGetData` but not immediately. Hence, `SQLFetch` does not have the information.

Because the ODBC driver guarantees data integrity, if you perform implicit bindings, redundant conversion may result in performance degradation. Your choice is the trade-off between performance with explicit binding or usability with implicit binding.

OLE DB Code Conversions

Unlike ODBC, OLE DB only enables you to perform implicit bindings for inserting, updating, and fetching data. The conversion algorithm for determining the intermediate character set is the same as the implicit binding cases of ODBC.

Table 6–8 OLE DB Implicit Bindings

Datatypes of OLE_ DB Client Buffer	Datatypes of the Target Column in the Database	In-Binding and Out-Binding Conversions	Comments
DBTYPE_WCHAR	CHAR, VARCHAR2, CLOB	If database character set is a subset of the NLS_LANG character set: Database character set to and from NLS_LANG character set in OCI. NLS_LANG character set to UTF-16 in OLE DB	No unexpected data loss May degrade performance if database character set is a subset of NLS_LANG character set
		If database character set is NOT a subset of NLS_LANG character set: Database character set to and from UTF-16 in OCI	
DBTYPE_CHAR	CHAR, VARCHAR2, CLOB	If database character set is a subset of the NLS_LANG character set: Database character set to and from NLS_LANG in OCI	No unexpected data loss May degrade performance if database character set is not a subset of NLS_LANG character set
		If database character set is not a subset of NLS_LANG character set: Database character set to and from UTF-16 in OCI. UTF-16 to NLS_ LANG character set in OLE DB	

ODBC Unicode Datatypes

In ODBC Unicode applications, use SQLWCHAR to store Unicode data. All standard Windows Unicode functions can be used for SQLWCHAR data manipulations. For example, wcslen counts the number of characters of SQLWCHAR data:

```
SQLWCHAR sqlStmt[] = L"select ename from emp";
len = wcslen(sqlStmt);
```

Microsoft's ODBC 3.5 specification defines three Unicode datatype identifiers for the SQL_C_WCHAR, SQL_C_WVARCHAR, and SQL_WLONGVARCHAR clients; and three Unicode datatype identifiers for servers SQL_WCHAR, SQL_WVARCHAR, and SQL_WLONGVARCHAR.

For binding operations, specify datatypes for both client and server using SQLBindParameter. The following is an example of Unicode binding, where the client buffer Name indicates that Unicode data (SQL_C_WCHAR) is bound to the first bind variable associated with the Unicode column (SQL_WCHAR):

```
SQLBindParameter(StatementHandle, 1, SQL_PARAM_INPUT, SQL_C_WCHAR,
SQL_WCHAR, NameLen, 0, (SQLPOINTER)Name, 0, &Name);
```

Table 6–9 represents the datatype mappings of the ODBC Unicode datatypes for the server against SQL NCHAR datatypes.

Table 6–9 Server ODBC Unicode Datatype Mapping

ODBC Datatype	Oracle Datatype
SQL_WCHAR	NCHAR
SQL_WVARCHAR	NVARCHAR2
SQL_WLONGVARCHAR	NCLOB

According to ODBC specifications, SQL_WCHAR, SQL_WVARCHAR, and SQL_WLONGVARCHAR are treated as Unicode data, and are therefore measured in the number of characters instead of the number of bytes.

OLE DB Unicode Datatypes

OLE DB offers the wchar_t *, BSTR, and OLESTR datatypes for the Unicode client C datatype. In practice, wchar_t is the most common datatype and the others are for specific purposes. The following example assigns a static SQL statement:

```
wchar_t *sqlStmt = OLESTR("SELECT ename FROM emp");
```

The OLESTR macro works exactly like an "L" modifier to indicate the Unicode string. If you need to allocate Unicode data buffer dynamically using OLESTR, use the IMalloc allocator (for example, CoTaskMemAlloc). However, using OLESTR is not the normal method for variable length data; use wchar_t* instead for generic string types. BSTR is similar. It is a string with a length prefix in the memory location preceding the string. Some functions and methods can accept only BSTR

Unicode datatypes. Therefore, BSTR Unicode string must be manipulated with special functions like `SysAllocString` for allocation and `SysFreeString` for freeing memory.

Unlike ODBC, OLE DB does not allow you to specify the server datatype explicitly. When you set the client datatype, the OLE DB driver automatically performs data conversion if necessary.

Table 6–10 illustrates OLE DB datatype mapping.

Table 6–10 OLE DB Datatype Mapping

OLE DB Datatype	Oracle Datatype
`DBTYPE_WCHAR`	`NCHAR or NVARCHAR2`

If DBTYPE_BSTR is specified, it is assumed to be DBTYPE_WCHAR because both are Unicode strings.

ADO Access

ADO is a high-level API to access database with the OLE DB and ODBC drivers. Most database application developers use the ADO interface on Windows because it is easily accessible from Visual Basic, the primary scripting language for Active Server Pages (ASP) for the Internet Information Server (IIS). To OLE DB and ODBC drivers, ADO is simply an OLE DB consumer or ODBC application. ADO assumes that OLE DB and ODBC drivers are Unicode-aware components; hence, it always attempts to manipulate Unicode data.

7 SQL and PL/SQL Programming in a Global Environment

This chapter contains information useful for SQL programming in a globalization support environment. It includes the following topics:

- Locale-Dependent SQL Functions with Optional NLS Parameters
- Other Locale-Dependent SQL Functions
- Miscellaneous Topics for SQL and PL/SQL Programming in a Global Environment

Locale-Dependent SQL Functions with Optional NLS Parameters

All SQL functions whose behavior depends on globalization support 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 enables the functions to be evaluated independently of the session's NLS parameters. This feature can be important for SQL statements that contain numbers and dates as string literals.

For example, the following query is evaluated correctly if the language specified for dates is AMERICAN:

```
SELECT last_name FROM employees WHERE hire_date > '01-JAN-1999';
```

Such a query can be made independent of the current date language by using a statement similar to the following:

```
SELECT last_name FROM employees WHERE hire_date >
TO_DATE('01-JAN-1999','DD-MON-YYYY', 'NLS_DATE_LANGUAGE = AMERICAN');
```

In this way, SQL statements that are independent of the session language can be defined where necessary. Such statements are necessary when string literals appear in SQL statements in views, CHECK constraints, or triggers.

All character functions support both single-byte and multibyte characters. Except where explicitly stated, character functions operate character by character, rather than byte by byte.

The rest of this section includes the following topics:

- Default Values for NLS Parameters in SQL Functions
- Specifying NLS Parameters in SQL Functions
- Unacceptable NLS Parameters in SQL Functions

Default Values for NLS Parameters in SQL Functions

When SQL functions evaluate views and triggers, default values from the current session are used for the NLS function parameters. When SQL functions evaluate `CHECK` constraints, they use the default values that were specified for the NLS parameters when the database was created.

Specifying NLS Parameters in SQL Functions

NLS parameters are specified in SQL functions as follows:

```
'parameter = value'
```

For example:

```
'NLS_DATE_LANGUAGE = AMERICAN'
```

The following NLS parameters can be specified in SQL functions:

- `NLS_DATE_LANGUAGE`
- `NLS_NUMERIC_CHARACTERS`
- `NLS_CURRENCY`
- `NLS_ISO_CURRENCY`
- `NLS_SORT`

Table 7–1 shows which NLS parameters are valid for specific SQL functions.

Table 7–1 SQL Functions and Their Valid NLS 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`

Table 7–1 SQL Functions and Their Valid NLS Parameters (Cont.)

SQL Function	Valid NLS Parameters
TO_NCHAR	NLS_DATE_LANGUAGE NLS_NUMERIC_CHARACTERS NLS_CURRENCY NLS_ISO_CURRENCY NLS_DUAL_CURRENCY NLS_CALENDAR
NLS_UPPER	NLS_SORT
NLS_LOWER	NLS_SORT
NLS_INITCAP	NLS_SORT
NLSSORT	NLS_SORT

The following examples show how to use NLS parameters in SQL functions:

```
TO_DATE ('1-JAN-99', 'DD-MON-YY',
   'nls_date_language = American')

TO_CHAR (hire_date, 'DD/MON/YYYY',
   'nls_date_language = French')

TO_NUMBER ('13.000,00', '99G999D99',
   'nls_numeric_characters = '',.''')

TO_CHAR (salary, '9G999D99L', 'nls_numeric_characters = '',.''
   nls_currency = '' Dfl''')

TO_CHAR (salary, '9G999D99C', 'nls_numeric_characters = ''.,''
   nls_iso_currency = Japan')

NLS_UPPER (last_name, 'nls_sort = Swiss')

NLSSORT (last_name, 'nls_sort = German')
```

Note: In some languages, some lowercase characters correspond to more than one uppercase character or vice versa. As a result, the length of the output from the NLS_UPPER, NLS_LOWER, and NLS_INITCAP functions can differ from the length of the input.

See Also: "Special Uppercase Letters" on page 4-12 and "Special Lowercase Letters" on page 4-12

Unacceptable NLS Parameters in SQL Functions

The following NLS parameters are not accepted in SQL functions except for NLSSORT:

- NLS_LANGUAGE
- NLS_TERRITORY
- NLS_DATE_FORMAT

NLS_DATE_FORMAT is not accepted as a parameter because it can interfere with required format masks. A date format must 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 the TO_CHAR or TO_DATE functions.

If NLS_LANGUAGE or NLS_TERRITORY is specified in the TO_CHAR, TO_NUMBER, or TO_DATE functions, then a format mask must also be specified as the second parameter of the function. For example, the following specification is legal:

```
TO_CHAR (hire_date, 'DD/MON/YYYY', 'nls_date_language = French')
```

The following specification is illegal because there is no format mask:

```
TO_CHAR (hire_date, 'nls_date_language = French')
```

The following specification is illegal because the format mask is not specified as the second parameter of the function:

```
TO_CHAR (hire_date, 'nls_date_language = French', 'DD/MON/YY')
```

Other Locale-Dependent SQL Functions

This section includes the following topics:

- The CONVERT Function
- SQL Functions for Different Length Semantics
- LIKE Conditions for Different Length Semantics
- Character Set SQL Functions
- The NLSSORT Function

The CONVERT Function

The CONVERT function enables 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 as conversion between database and client character sets. Hence, it uses replacement characters and has the same limitations.

> **See Also:** "Character Set Conversion Between Clients and the Server" on page 2-16

The syntax for CONVERT is as follows:

```
CONVERT(char, dest_char_set[, source_char_set])
```

source_char_set is the source character set and dest_char_set is the destination character set. If the source_char_set parameter is not specified, then it defaults to the database character set.

In client/server environments that use different character sets, use the TRANSLATE ...USING function to perform conversions instead of CONVERT. The TRANSLATE...USING function must be used if either the client or the server has NCHAR or NVARCHAR2 data.

> **See Also:**
>
> - *Oracle9i SQL Reference* for more information about the CONVERT function and the TRANSLATE...USING function
> - "Character Set Conversion Support" on page A-18 for character set encodings that are used only for the CONVERT function

SQL Functions for Different Length Semantics

Oracle9*i* provides SQL functions that work in accordance with different length semantics. There are three groups of such SQL functions: SUBSTR, LENGTH, and INSTR. Each function in a group is based on a different kind of length semantics and is distinguished by the character or number appended to the function name.The members of each group of functions is distinguished by the character or number that is appended to the function's name. For example, SUBSTRB is based on byte semantics.

The SUBSTR functions return a requested portion of a substring. The LENGTH functions return the length of a string. The INSTR functions search for a substring in a string.

The SUBSTR functions calculate the length of a string differently. Table 7–1 summarizes the calculation methods.

Table 7–2 How the SUBSTR Functions Calculate the Length of a String

Function	Calculation Method
SUBSTR	Calculates the length of a string in characters based on the length semantics associated with the character set of the datatype. For example, AL32UTF8 characters are calculated in UCS-4 code units. UTF8 and AL16UTF16 characters are calculated in UCS-2 code units. A supplementary character is counted as one character in AL32UTF8 and as two characters in UTF8 and AL16UTF16. Because VARCHAR and NVARCHAR may use different character sets, SUBSTR may give different results for different datatypes even if two strings are identical. If your application requires consistency, then use SUBSTR2 or SUBSTR4 to force all semantic calculations to be UCS-2 or UCS-4, respectively.
SUBSTRB	Calculates the length of a string in bytes
SUBSTR2	Calculates the length of a string in UCS-2 code units, which is compliant with Java strings and Windows client environments. Characters are represented in UCS-2 or 16-bit Unicode values. Supplementary characters are counted as two code units.
SUBSTR4	Calculates the length of a string in UCS-4 code units. Characters are represented in UCS-4 or 32-bit Unicode values. Supplementary characters are counted as one code unit.
SUBSTRC	Calculates the length of a string in Unicode complete characters. Supplementary characters and composite characters are counted as one character.

The LENGTH and INSTR functions calculate string length in the same way, according to the character or number added to the function name.

The following examples demonstrate the differences between SUBSTR and SUBSTRB on a database whose character set is AL32UTF8.

For the string Fußball, the following statement returns a substring that is 4 characters long, beginning with the second character:

```
SELECT SUBSTR ('Fußball', 2 , 4) SUBSTR FROM dual;
```

```
SUBS
----
ußba
```

For the string Fußball, the following statement returns a substring 4 bytes long, beginning with the second byte:

```
SELECT SUBSTRB ('Fußball', 2 , 4) SUBSTRB FROM dual;

SUB
---
ußb
```

See Also: *Oracle9i SQL Reference* for more information about the SUBSTR, LENGTH, and INSTR functions

LIKE Conditions for Different Length Semantics

The LIKE conditions specify a test that uses pattern-matching. The equality operator (=) exactly matches one character value to another, but the LIKE conditions match a portion of one character value to another by searching the first value for the pattern specified by the second.

LIKE calculates the length of strings in characters using the length semantics associated with the input character set. The LIKE2, LIKE4, and LIKEC conditions are summarized in Table 7–3.

Table 7–3 LIKE Conditions

Function	Description
LIKE2	Use when characters are represented in UCS-2 semantics. A supplementary character is considered as two code units.
LIKE4	Use when characters are represented in UCS-4 semantics. A supplementary character is considered as one code unit.
LIKEC	Use when characters are represented in Unicode complete character semantics. A composed character is treated as one code unit.

There is no LIKEB condition.

Character Set SQL Functions

Two SQL functions, `NLS_CHARSET_NAME` and `NLS_CHARSET_ID`, can 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.

Another SQL function, `NLS_CHARSET_DECL_LEN`, returns the length of an `NCHAR` column.

This section includes the following topics:

- Converting from Character Set Number to Character Set Name
- Converting from Character Set Name to Character Set Number
- Returning the Length of an NCHAR Column

See Also: *Oracle9i SQL Reference*

Converting from Character Set Number to Character Set Name

The `NLS_CHARSET_NAME(`*`n`*`)` function returns the name of the character set corresponding to ID number *`n`*. The function returns `NULL` if *`n`* is not a recognized character set ID value.

Converting from Character Set Name to Character Set Number

`NLS_CHARSET_ID(`*`text`*`)` returns the character set ID corresponding to the name specified by *`text`*. *`text`* is defined as a run-time `VARCHAR2` quantity, a character set name. Values for `text` can be `NLSRTL` names that resolve to character sets that are not the database character set or the national character set.

If the value `CHAR_CS` is entered for *`text`*, then the function returns the ID of the server's database character set. If the value `NCHAR_CS` is entered for *`text`*, then the function returns the ID of the server's national character set. The function returns `NULL` if *`text`* is not a recognized name.

Note: The value for *`text`* must be entered in uppercase characters.

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. *`BYTECNT`* is the byte length of the column. *`CSID`* is the character set ID of the column.

The NLSSORT Function

The `NLSSORT` function enables you to use any linguistic sort for an `ORDER BY` clause. It replaces a character string with the equivalent sort string used by the linguistic sort mechanism so that sorting the replacement strings produces the desired sorting sequence. For a binary sort, the sort string is the same as the input string.

The kind of linguistic sort used by an `ORDER BY` clause is determined by the `NLS_SORT` session parameter, but it can be overridden by explicitly using the `NLSSORT` function.

Example 7–1 specifies a German sort with the `NLS_SORT` session parameter.

Example 7–1 Specifying a German Sort with the NLS_SORT Session Parameter

```
ALTER SESSION SET NLS_SORT = GERMAN;
SELECT * FROM table1
ORDER BY column1;
```

Example 7–2 Specifying a French Sort with the NLSSORT Function

This example first sets the `NLS_SORT` session parameter to German, but the `NLSSORT` function overrides it by specifying a French sort.

```
ALTER SESSION SET NLS_SORT = GERMAN;
SELECT * FROM table1
ORDER BY NLSSORT(column1, 'NLS_SORT=FRENCH');
```

The `WHERE` clause uses binary comparison rather than linguistic comparison by default, but this can be overridden by using the `NLSSORT` function in the `WHERE` clause.

Example 7–3 Making a Linguistic Comparison with the WHERE Clause

```
ALTER SESSION SET NLS_COMP = ANSI;
SELECT * FROM table1
WHERE NLSSORT(column1, 'NLS_SORT=FRENCH')>
    NLSSORT(column2, 'NLS_SORT=FRENCH');
```

Setting the NLS_COMP session parameter to ANSI causes the NLS_SORT value to be used in the WHERE clause.

The rest of this section contains the following topics:

- NLSSORT Syntax
- Comparing Strings in a WHERE Clause
- Using the NLS_COMP Parameter to Simplify Comparisons in the WHERE Clause
- Controlling an ORDER BY Clause

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 NLSSORT function is not the same as the NLS_ LANG client environment setting. In the NLSSORT function, NLS_LANG specifies the abbreviated language name, such as US for American or PL for Polish. For example:

```
SELECT * FROM table1
ORDER BY NLSSORT(column1, 'NLS_LANG=PL');
```

Comparing Strings in a WHERE Clause

NLSSORT enables applications to perform string matching that follows alphabetic conventions. Normally, character strings in a WHERE clause are compared by using the binary values of the characters. One character is considered greater than another character if it has a greater 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 may not follow alphabetic conventions. For example, if a column (`column1`) contains the values ABC, ABZ, BCD, and ÄBC in the ISO 8859-1 8-bit character set, the following query returns both BCD and ÄBC because Ä has a higher numeric value than B:

```
SELECT column1 FROM table1 WHERE column1 > 'B';
```

In German, Ä is sorted alphabetically before B, but in Swedish, Ä is sorted after Z. Linguistic comparisons can be made by using NLSSORT in the WHERE clause:

```
WHERE NLSSORT(col) comparison_operator NLSSORT(comparison_string)
```

Note that NLSSORT must be on both sides of the comparison operator. For example:

```
SELECT column1 FROM table1 WHERE NLSSORT(column1) > NLSSORT('B');
```

If a German linguistic sort has been set, then the statement does not return strings beginning with Ä because Ä comes before B in the German alphabet. If a Swedish linguistic sort has been set, then strings beginning with Ä are returned because Ä comes after Z in the Swedish alphabet.

Using the NLS_COMP Parameter to Simplify Comparisons in the WHERE Clause

Comparison in the WHERE clause or PL/SQL blocks is binary by default. Using the NLSSORT function for linguistic comparison can be tedious, especially when the linguistic sort has already been specified in the NLS_SORT session parameter. You can use the NLS_COMP parameter to indicate that the comparisons in a WHERE clause or in PL/SQL blocks must be linguistic according to the NLS_SORT session parameter.

Note: The NLS_COMP parameter does not affect comparison behavior for partitioned tables. String comparisons that are based on a VALUES LESS THAN partition are always binary.

See Also: "NLS_COMP" on page 3-41

Controlling an ORDER BY Clause

If a linguistic sort is in use, then ORDER BY clauses use an implicit NLSSORT on character data. The sort mechanism (linguistic or binary) for an ORDER BY clause is transparent to the application. However, if the NLSSORT function is explicitly specified in an ORDER BY clause, then the implicit NLSSORT is not done.

If a linguistic sort has been defined by the NLS_SORT session parameter, then an ORDER BY clause in an application uses an implicit NLSSORT function. If you specify an explicit NLSSORT function, then it overrides the implicit NLSSORT function.

When the sort mechanism has been defined as linguistic, the NLSSORT function is usually unnecessary in an ORDER BY clause.

When the sort mechanism either defaults or is defined as binary, then a query like the following uses a binary sort:

```
SELECT last_name FROM employees
ORDER BY last_name;
```

A German linguistic sort can be obtained as follows:

```
SELECT last_name FROM employees
ORDER BY NLSSORT(last_name, 'NLS_SORT = GERMAN');
```

See Also: "Improving Case-Insensitive Searches with a Function-Based Index" on page 4-16

Miscellaneous Topics for SQL and PL/SQL Programming in a Global Environment

This section contains the following topics:

- SQL Date Format Masks
- Calculating Week Numbers
- SQL Numeric Format Masks
- The Concatenation Operator
- Loading External BFILE Data into LOBs

See Also: *Oracle9i SQL Reference* for a complete description of format masks

SQL Date Format Masks

Several format masks are provided with the `TO_CHAR`, `TO_DATE`, and `TO_NUMBER` functions.

The `RM` (Roman Month) format element returns a month as a Roman numeral. You can specify either upper case or lower case by using `RM` or `rm`. For example, for the date 7 Sep 1998, `DD-rm-YYYY` returns `07-ix-1998` and `DD-RM-YYYY` returns `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.

Calculating Week Numbers

The week numbers returned by the `WW` format mask are calculated according to the following algorithm: `int(dayOfYear+6)/7`. This algorithm does not follow the ISO standard (2015, 1992-06-15).

To support the ISO standard, the `IW` format element is provided. It returns the ISO week number. In addition, the `I`, `IY`, `IYY`, and `IYYY` format elements, equivalent in behavior to the Y, YY, YYY, and YYYY format elements, 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. The week number is determined according the following rules:

- 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 in week 1. Thus, the ISO week number and year for December 31, 1990, is 1, 1991. To get the ISO week number, use the `IW` format mask for the week number and one of the `IY` formats for the year.

SQL Numeric Format Masks

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, you can specify either upper case or lower case, using `RN` or `rn`, respectively. The number being converted must be an integer in the range 1 to 3999.

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 (encoded as ASCII 124) will fail. For example, creating a procedure fails 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.

The user 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.

Loading External BFILE Data into LOBs

The `DBMS_LOB` PL/SQL package can load external `BFILE` data into `LOB`s. Previous releases of Oracle did not perform character set conversion before loading the binary data into `CLOB`s or `NCLOB`s. Thus the `BFILE` data had to be in the same character set as the database or national character set to work properly. The APIs that are introduced in Oracle9*i* Release 2 (9.2) allow the user to specify the character set ID of the `BFILE` data by using a new parameter. The APIs convert the data from the specified `BFILE` character set into the database character set for `CLOB`s or the national character set for `NCLOB`s. The loading takes place on the server because `BFILE` data is not supported on the client.

- Use `DBMS_LOB.LOADBLOBFROMFILE` to load to `BLOB`s.
- Use `DBMS_LOB.LOADCLOBFROMFILE` for load to `CLOB`s and `NCLOB`s.

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)*

8

OCI Programming in a Global Environment

This chapter contains information useful for OCI programming. It includes the following topics:

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 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 an `ALTER SESSION` statement. The character set associated with the session handle is the database character set.

Note that the OCI session handle does not have any NLS settings associated with it until the first transaction begins in the session. `SELECT` statements do not begin a transaction.

Specifying Character Sets in OCI

Use the `OCIEnvNlsCreate` function to specify client-side database and national character sets when the OCI environment is created.This function allows users to set character set information dynamically in applications, independent of the `NLS_LANG` and `NLS_CHAR` initialization parameter settings. In addition, one application can initialize several environment handles for different client environments in the same server environment.

Any Oracle character set ID except AL16UTF16 can be specified through the `OCIEnvNlsCreate` function to specify the encoding of metadata, SQL `CHAR` data, and SQL `NCHAR` data. Use `OCI_UTF16ID` in the `OCIEnvNlsCreate` function, introduced in Oracle 9*i* Release 2 (9.2), to specify UTF-16 data. Note that the `OCI_UTF16` parameter in the `OCIEnvCreate` function, which was introduced in Oracle9*i* release 1 (9.0.1) and was known as Unicode mode, has been deprecated.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about the `OCIEnvNlsCreate` function and the `OCIEnvCreate` function

OCIEnvNlsCreate()

Syntax

```
sword OCIEnvNlsCreate   ( OCIEnv        **envhpp,
                          ub4           mode,
                          dvoid         *ctxp,
dvoid         *(*malocfp)
                   (dvoid *ctxp,
                    size_t size),
dvoid         *(*ralocfp)
                   (dvoid *ctxp,
                    dvoid *memptr,
                    size_t newsize),
void          (*mfreefp)
                   (dvoid *ctxp,
                    dvoid *memptr))
size_t        xtramemsz,
dvoid         **usrmempp
ub2           charset,
ub2           ncharset );
```

Purpose

Creates and initializes an environment handle for OCI functions to work under. It is an enhanced version of the `OCIEnvCreate()` function.

Parameters

envhpp (OUT)

A pointer to an environment handle whose encoding setting is specified by *`mode`*. The setting is inherited by statement handles derived from *`envhpp`*.

mode (IN)

Specifies initialization of the mode. Valid modes are:

- `OCI_DEFAULT`: The default value, which is non-UTF-16 encoding.
- `OCI_THREADED`: Uses threaded environment. Internal data structures not exposed to the user are protected from concurrent accesses by multiple threads.
- `OCI_OBJECT`: Uses object features.
- `OCI_UTF16`: The environment handle and handles inherited from it assume UTF-16 encoding. This setting is deprecated. Instead, specify `OCI_UTF16ID` for both *`charset`* and *`ncharset`*.
- `OCI_SHARED`: Uses shared data structures.
- `OCI_EVENTS`: Uses publish-subscribe notifications.

- `OCI_NO_UCB`: Suppresses the calling of the *`OCIEnvCallback`* dynamic callback routine. The default behavior is to allow calling of *`OCIEnvCallback`* at the time that the environment is created.
- `OCI_ENV_NO_MUTEX`: No mutexing in this mode. All OCI calls done on the environment handle, or on handles derived from the environment handle, must be serialized.

ctxp (IN)
Specifies the user-defined context for the memory callback routines.

malocfp (IN)
Specifies the user-defined memory allocation function. If the mode is `OCI_THREADED`, then this memory allocation routine must be thread-safe.

ctxp (IN)
Specifies the context pointer for the user-defined memory allocation function.

size (IN)
Specifies the size of memory to be allocated by the user-defined memory allocation function.

ralocfp (IN)
Specifies the user-defined memory re-allocation function. If the mode is `OCI_THREADED`, then this memory allocation routine must be thread-safe.

ctxp (IN)
Specifies the context pointer for the user-defined memory reallocation function.

memp (IN)
Pointer to memory block.

newsize (IN)
Specifies the new size of memory to be allocated

mfreefp (IN)
Specifies the user-defined memory free function. If the mode is `OCI_THREADED`, then this memory-free routine must be thread-safe.

ctxp (IN)
Specifies the context pointer for the user-defined memory-free function.

memptr (IN)
Pointer to memory to be freed

xtramemsz (IN)
Specifies the amount of user memory to be allocated for the duration of the environment.

usrmempp (OUT)
Returns a pointer to the user memory of size *`xtramemsz`* allocated by the call for the user.

charset (IN)
The client-side character set for the current environment handle. If it is `0`, then the `NLS_LANG` setting is used. `OCI_UTF16ID` is a valid setting. This affects metadata and `CHAR` data.

ncharset (IN)
The client-side national character set for the current environment handle. If it is `0`, then the `NLS_NCHAR` setting is used. `OCI_UTF16ID` is a valid setting. This affects `NCHAR` data.

Returns

`OCI_SUCCESS`: The environment handle has been successfully created.

`OCI_ERROR`: An error occurred.

Comments

Note: This call should be invoked before any other OCI call and should be used instead of the `OCIInitialize()` and `OCIEnvInit()` calls. `OCIInitialize()` and `OCIEnvInit()` calls are supported for backward compatibility.

This function sets nonzero *`charset`* and *`ncharset`* as client-side database and national character sets, replacing the ones specified by `NLS_LANG` and `NLS_NCHAR`. When *`charset`* and *`ncharset`* are 0, it behaves exactly the same as `OCIEnvCreate()`. Specifically, *`charset`* controls the encoding for metadata and data with implicit form attribute and *`ncharset`* controls the encoding for data with `SQLCS_NCHAR` form attribute.

Although `OCI_UTF16ID` can be set by `OCIEnvNlsCreate()`, `NLS_LANG` and `NLS_NCHAR` cannot have a UTF-16 setting.

The character set IDs in `NLS_LANG` and `NLS_NCHAR` can be retrieved with `OCINlsEnvironmentVariableGet()`.

This call returns an environment handle which is then used by the remaining OCI functions. There can be multiple environments in OCI, each with its own environment modes. This function also performs any process-level initialization if required by any mode. For example, if the user wants to initialize an environment as OCI_THREADED, then all libraries that are used by OCI are also initialized in the threaded mode.

If you are writing a DLL or a shared library using OCI library then this call should be used instead of `OCIInitialize()` and `OCIEnvInit()` calls.

See Also: "OCINlsEnvironmentVariableGet()" on page 8-13

Getting Locale Information in OCI

An Oracle locale consists of language, territory, and character set definitions. The locale determines conventions such as day and month names, as well as date, time, number, and currency formats. A globalized application obeys a user's locale setting and cultural conventions. For example, when the locale is set to German, users expect to see day and month names in German.

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

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

This section includes the following topics:

- OCINlsGetInfo()
- OCI_NLS_MAXBUFSZ
- Example: Getting Locale Information in OCI
- OCINlsCharSetNameTold()
- OCINlsCharSetIdToName()
- OCINlsNumericInfoGet()
- OCINlsEnvironmentVariableGet()

OCINlsGetInfo()

Syntax

```
sword OCINlsGetInfo(dvoid *hndl, OCIError *errhp, OraText *buf, size_t buflen,
ub2 item)
```

Purpose

This function obtains locale information specified by `item` from an OCI environment or user session handle (`hndl`) into an array pointed to by `buf` within a size limitation specified by `buflen`.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`

Parameters

hndl(IN/OUT)
The OCI environment or user session handle initialized in object mode

errhp(IN/OUT)
The OCI error handle. If there is an error, then it is recorded in `errhp` and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.

buf(OUT)
Pointer to the destination buffer. Returned strings are terminated by a `NULL` character.

buflen(IN)
The size of the destination buffer. The maximum length for each piece of information is `OCI_NLS_MAXBUFSZ` bytes

item(IN)
Specifies which item in the OCI environment handle to return. It 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
`OCI_NLS_MONTHNAME1`: Native name for January
`OCI_NLS_MONTHNAME2`: Native name for February
`OCI_NLS_MONTHNAME3`: Native name for March
`OCI_NLS_MONTHNAME4`: Native name for April
`OCI_NLS_MONTHNAME5`: Native name for May
`OCI_NLS_MONTHNAME6`: Native name for June
`OCI_NLS_MONTHNAME7`: Native name for July
`OCI_NLS_MONTHNAME8`: Native name for August
`OCI_NLS_MONTHNAME9`: Native name for September
`OCI_NLS_MONTHNAME10`: Native name for October
`OCI_NLS_MONTHNAME11`: Native name for November
`OCI_NLS_MONTHNAME12`: Native name for December
`OCI_NLS_ABMONTHNAME1`: Native abbreviated name for January
`OCI_NLS_ABMONTHNAME2`: Native abbreviated name for February
`OCI_NLS_ABMONTHNAME3`: Native abbreviated name for March
`OCI_NLS_ABMONTHNAME4`: Native abbreviated name for April
`OCI_NLS_ABMONTHNAME5`: Native abbreviated name for May
`OCI_NLS_ABMONTHNAME6`: Native abbreviated name for June
`OCI_NLS_ABMONTHNAME7`: Native abbreviated name for July
`OCI_NLS_ABMONTHNAME8`: Native abbreviated name for August

`OCI_NLS_ABMONTHNAME9`: Native abbreviated name for September
`OCI_NLS_ABMONTHNAME10`: Native abbreviated name for October
`OCI_NLS_ABMONTHNAME11`: Native abbreviated name for November
`OCI_NLS_ABMONTHNAME12`: Native abbreviated name for December
`OCI_NLS_YES`: Native string for affirmative response
`OCI_NLS_NO`: Native negative response
`OCI_NLS_AM`: Native equivalent string of AM
`OCI_NLS_PM`: Native equivalent string of PM
`OCI_NLS_AD`: Native equivalent string of AD
`OCI_NLS_BC`: Native equivalent string of BC
`OCI_NLS_DECIMAL`: Decimal character
`OCI_NLS_GROUP`: Group separator
`OCI_NLS_DEBIT`: Native symbol of debit
`OCI_NLS_CREDIT`: Native symbol of credit
`OCI_NLS_DATEFORMAT`: Oracle date format
`OCI_NLS_INT_CURRENCY`: International currency symbol
`OCI_NLS_DUAL_CURRENCY`: Dual currency symbol
`OCI_NLS_LOC_CURRENCY`: Locale currency symbol
`OCI_NLS_LANGUAGE`: Language name
`OCI_NLS_ABLANGUAGE`: Abbreviation for language name
`OCI_NLS_TERRITORY`: Territory name
`OCI_NLS_CHARACTER_SET`: Character set name
`OCI_NLS_LINGUISTIC_NAME`: Linguistic sort name
`OCI_NLS_CALENDAR`: Calendar name
`OCI_NLS_WRITING_DIR`: Language writing direction
`OCI_NLS_ABTERRITORY`: Territory abbreviation
`OCI_NLS_DDATEFORMAT`: Oracle default date format
`OCI_NLS_DTIMEFORMAT`: Oracle default time format
`OCI_NLS_SFDATEFORMAT`: Local date format
`OCI_NLS_SFTIMEFORMAT`: Local time format
`OCI_NLS_NUMGROUPING`: Number grouping fields
`OCI_NLS_LISTSEP`: List separator
`OCI_NLS_MONDECIMAL`: Monetary decimal character
`OCI_NLS_MONGROUP`: Monetary group separator
`OCI_NLS_MONGROUPING`: Monetary grouping fields
`OCI_NLS_INT_CURRENCYSEP`: International currency separator

OCI_NLS_MAXBUFSZ

When calling `OCINlsGetInfo`(), you need to allocate the buffer to store the returned information. The buffer size depends on which item you are querying and what encoding you are using to store the information. Developers should not need to know how many bytes it takes to store `January` in Japanese using JA16SJIS encoding. The `OCI_NLS_MAXBUFSZ` attribute guarantees that the buffer is big enough to hold the largest item returned by `OCINlsGetInfo`().

Example: Getting Locale Information in OCI

This example code retrieves information and checks 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);
}
```

OCINlsCharSetNameToId()

Syntax

```
 ub2 OCINlsCharSetNameToId(dvoid *hndl, const oratext *name)
```

Purpose

This function returns the Oracle character set ID for the specified Oracle character set name.

Returns

Character set ID if the specified character set name and the OCI handle are valid. Otherwise it returns `0`.

Parameters

hndl(IN/OUT)

OCI environment or session handle. If the handle is invalid, then the function returns zero.

name(IN)

Pointer to a null-terminated Oracle character set name. If the character set name is invalid, then the function returns zero.

OCINlsCharSetIdToName()

Syntax

```
sword OCINlsCharSetIdToName( dvoid *hndl, oratext *buf, size_t buflen, ub2 id)
```

Purpose

This function returns the Oracle character set name from the specified character set ID.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`

Parameters

hndl(IN/OUT)

OCI environment or session handle. If the handle is invalid, then the function returns `OCI_INVALID_HANDLE`.

buf(OUT)

Points to the destination buffer. If the function returns `OCI_SUCCESS`, then the parameter contains a null-terminated string for the character set name.

buflen(IN)

The size of the destination buffer. The recommended size is `OCI_NLS_MAXBUFSZ` to guarantee storage for an Oracle character set name. If the size of the destination buffer is smaller than the length of the character set name, the function returns `OCI_ERROR`.

id(IN)
Oracle character set ID

OCINlsNumericInfoGet()

Syntax

```
sword OCINlsNumericInfoGet( dvoid *hndl, OCIError *errhp, sb4 *val, ub2 item)
```

Purpose

This function obtains numeric language information specified by `item` from the OCI environment handle into an output number variable.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`

Parameters

hndl(IN/OUT)
OCI environment or session handle. If the handle is invalid, then the function returns `OCI_INVALID_HANDLE`.

errhp(IN/OUT)
The OCI error handle. If there is an error, then it is recorded in `errhp` and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.

val(OUT)
Pointer to the output number variable. If the function returns `OCI_SUCCESS`, then the parameter contains the requested NLS numeric information.

item(IN)
It specifies which item to get from the OCI environment handle and can be one of following values:

- `OCI_NLS_CHARSET_MAXBYTESZ`: Maximum character byte size for OCI environment or session handle character set

- OCI_NLS_CHARSET_FIXEDWIDTH: Character byte size for fixed-width character set; 0 for variable-width character set

OCINlsEnvironmentVariableGet()

Purpose

Returns the character set ID from NLS_LANG or the national character set id from NLS_NCHAR.

Syntax

```
sword OCINlsEnvironmentVariableGet ( dvoid     *val,
                                     size_t    size,
                                     ub2       item,
                                     ub2 charset,
                                     size_t    *rsize );
```

Parameters

val (IN/OUT)
Returns a value of an NLS environment variable such as the NLS_LANG character set ID or the NLS_NCHAR character set ID

size (IN)
Specifies the size of the given output value, which is applicable only to string data. The maximum length for each piece of information is OCI_NLS_MAXBUFSZ bytes. In the case of numeric data, this argument is ignored.

item (IN)
Specifies one of the following values to get from the NLS environment variable:

- OCI_NLS_CHARSET_ID: NLS_LANG character set ID in ub2 datatype
- OCI_NLS_NCHARSET_ID: NLS_NCHAR character set ID in ub2 datatype

charset (IN)
Specifies the character set ID for retrieved string data. If it is 0, then the NLS_LANG value is used. OCI_UTF16ID is a valid value for this argument. In the case of numeric data, this argument is ignored.

rsize (OUT)
The length of the return value in bytes

Returns

`OCI_SUCCESS`: The function finished successfully.

`OCI_ERROR`: An error occurred.

Comments

Following NLS convention, the national character set ID is the same as the character set ID if `NLS_NCHAR` is not set. If `NLS_LANG` is not set, tn the default character set ID is returned.

To allow for future enhancements of this function (to retrieve other values from environment variables) the datatype of the output `val` is a pointer to `dvoid`. String data is not terminated by `NULL`.

Note that the function does not take an environment handle, so the character set ID and the national character set ID that it returns are the values specified in `NLS_LANG` and `NLS_NCHAR`, instead of the values saved in the OCI environment handle. To get the character set IDs used by the OCI environment handle, call `OCIAttrGet()` for `OCI_ATTR_ENV_CHARSET` and `OCI_ATTR_ENV_NCHARSET`.

Mapping Locale Information Between Oracle and Other Standards

The `OCINlsNameMap` function maps Oracle character set names, language names, and territory names to and from Internet Assigned Numbers Authority (IANA) and International Organization for Standardization (ISO) names.

OCINlsNameMap()

Syntax

```
sword OCINlsNameMap( dvoid *hndl, oratext *buf, size_t buflen, const oratext
*srcbuf, uword flag)
```

Purpose

This function maps Oracle character set names, language names, and territory names to and from IANA and ISO names.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`

Parameters

hndl(IN/OUT)
OCI environment or session handle. If the handle is invalid, then the function returns `OCI_INVALID_HANDLE`.

buf(OUT)
Points to the destination buffer. If the function returns `OCI_SUCCESS`, then the parameter contains a null-terminated string for the requested name.

buflen(IN)
The size of the destination buffer. The recommended size is `OCI_NLS_MAXBUFSZ` to guarantee storage of an NLS name. If the size of the destination buffer is smaller than the length of the name, then the function returns `OCI_ERROR`.

srcbuf(IN)
Pointer to a null-terminated NLS name. If it is not a valid name, then the function returns `OCI_ERROR`.

flag(IN)
It specifies the direction of the name mapping and can take the following values:

`OCI_NLS_CS_IANA_TO_ORA`: Map character set name from IANA to Oracle
`OCI_NLS_CS_ORA_TO_IANA`: Map character set name from Oracle to IANA.
`OCI_NLS_LANG_ISO_TO_ORA`: Map language name from ISO to Oracle
`OCI_NLS_LANG_ORA_TO_ISO`: Map language name from Oracle to ISO
`OCI_NLS_TERR_ISO_TO_ORA`: Map territory name from ISO to Oracle
`OCI_NLS_TERR_ORA_TO_ISO`: Map territory name from Oracle to ISO
`OCI_NLS_TERR_ISO3_TO_ORA`: Map territory name from 3-letter ISO abbreviation to Oracle
`OCI_NLS_TERR_ORA_TO_ISO3`: Map territory name from Oracle to 3-letter ISO abbreviation

Manipulating Strings in OCI

Two types of data structures are supported for string manipulation:

- Multibyte strings
- Wide character strings

Multibyte strings are encoded in native Oracle character sets. Functions that operate on multibyte strings take the string as a whole unit with the length of the string calculated in bytes. Wide character (`wchar`) string functions provide more flexibility in string manipulation. They support character-based and string-based operations with the length the string calculated in characters.

The wide character datatype is Oracle-specific and should not be confused with the `wchar_t` datatype defined by the ANSI/ISO C standard. The Oracle wide character datatype is always 4 bytes in all platforms, while the size of `wchar_t` depends on the implementation and the platform. The Oracle wide character datatype normalizes multibyte characters so that they have a fixed width for easy processing. This guarantees no data loss for round-trip conversion between the Oracle wide character set and the native character set.

String manipulation can be classified into the following categories:

- Conversion of strings between multibyte and wide character
- Character classifications
- Case conversion
- Calculations of display length
- General string manipulation, such as comparison, concatenation, and searching

Table 8–1 summarizes the OCI string manipulation functions. They are described in more detail in the rest of this section.

Table 8–1 OCI String Manipulation Functions

Function	Description
`OCIMultiByteToWideChar()`	Converts an entire null-terminated string into the `wchar` format
`OCIMultiByteInSizeToWideChar()`	Converts part of a string into the `wchar` format
`OCIWideCharToMultiByte()`	Converts an entire null-terminated wide character string into a multibyte string
`OCIWideCharInSizeToMultiByte()`	Converts part of a wide character string into the multibyte format
`OCIWideCharToLower()`	Converts the `wchar` character specified by `wc` into the corresponding lowercase character if it exists in the specified locale. If no corresponding lowercase character exists, then it returns `wc` itself.
`OCIWideCharToUpper()`	Converts the `wchar` character specified by `wc` into the corresponding uppercase character if it exists in the specified locale. If no corresponding uppercase character exists, then it returns `wc` itself.
`OCIWideCharStrcmp()`	Compares two wide character strings by binary, linguistic, or case-insensitive comparison method

Table 8–1 OCI String Manipulation Functions (Cont.)

Function	Description
`OCIWideCharStrncmp()`	Similar to `OCIWideCharStrcmp()`. Compares two wide character strings by binary, linguistic, or case-insensitive comparison methods. At most `len1` bytes form `str1`, and `len2` bytes form `str2`.
`OCIWideCharStrcat()`	Appends a copy of the string pointed to by `wsrcstr`. Then it returns the number of characters in the resulting string.
`OCIWideCharStrncat()`	Appends a copy of the string pointed to by `wsrcstr`. Then it returns the number of characters in the resulting string. At most *n* characters are appended.
`OCIWideCharStrchr()`	Searches for the first occurrence of `wc` in the string pointed to by `wstr`. Then it returns a pointer to the `wchar` if the search is successful.
`OCIWideCharStrrchr()`	Searches for the last occurrence of `wc` in the string pointed to by `wstr`
`OCIWideCharStrcpy()`	Copies the `wchar` string pointed to by `wsrcstr` into the array pointed to by `wdststr`. Then it returns the number of characters copied.
`OCIWideCharStrncpy()`	Copies the `wchar` string pointed to by `wsrcstr` into the array pointed to by `wdststr`. Then it returns the number of characters copied. At most *n* characters are copied from the array.
`OCIWideCharStrlen()`	Computes the number of characters in the `wchar` string pointed to by `wstr` and returns this number
`OCIWideCharStrCaseConversion()`	Converts the wide character string pointed to by `wsrcstr` into the case specified by a flag and copies the result into the array pointed to by `wdststr`
`OCIWideCharDisplayLength()`	Determines the number of column positions required for `wc` in display
`OCIWideCharMultibyteLength()`	Determines the number of bytes required for `wc` in multibyte encoding
`OCIMultiByteStrcmp()`	Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods
`OCIMultiByteStrncmp()`	Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods. At most `len1` bytes form `str1` and `len2` bytes form `str2.`
`OCIMultiByteStrcat()`	Appends a copy of the multibyte string pointed to by `srcstr`
`OCIMultiByteStrncat()`	Appends a copy of the multibyte string pointed to by `srcstr`. At most *n* bytes from `srcstr` are appended to `dststr`

Table 8–1 OCI String Manipulation Functions (Cont.)

Function	Description
`OCIMultiByteStrcpy()`	Copies the multibyte string pointed to by `srcstr` into an array pointed to by `dststr`. It returns the number of bytes copied.
`OCIMultiByteStrncpy()`	Copies the multibyte string pointed to by `srcstr` into an array pointed to by `dststr`. It returns the number of bytes copied. At most *n* bytes are copied from the array pointed to by `srcstr` to the array pointed to by `dststr`.
`OCIMultiByteStrlen()`	Returns the number of bytes in the multibyte string pointed to by `str`
`OCIMultiByteStrnDisplayLength()`	Returns the number of display positions occupied by the complete characters within the range of *n* bytes
`OCIMultiByteStrCaseConversion()`	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);
```

Purpose

This routine converts an entire `NULL`-terminated string into the `wchar` format. The `wchar` output buffer are `NULL`-terminated. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE` or `OCI_ERROR`

Parameters

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)
```

Purpose
This routine converts part of a string into the wchar format. It converts as many complete characters as it can until it reaches the output buffer size limit or input buffer size limit or it reaches a NULL terminator in a source string. The output buffer is NULL-terminated if space permits. If dstsz is zero, then this function returns only the number of characters not including the ending NULL terminator needed for a converted string. If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate function, then this function produces an error.

Returns
OCI_SUCCESS, OCI_INVALID_HANDLE or OCI_ERROR

Parameters

hndl(IN/OUT)
OCI environment or user session handle to determine the character set of the string

dst(OUT)
Pointer to a destination buffer for wchar. It can be NULL pointer when dstsz is zero.

dstsz(IN)
Destination buffer size in number of characters. If it is zero, this function just returns number of characters needed for the conversion.

src (IN)
Source string to be converted

srcsz(IN)
Length of source string in bytes

rsize(OUT)
Number of characters written into destination buffer, or number of characters for converted string if `dstsz` is zero. If it is a `NULL` pointer, nothing is returned.

OCIWideCharToMultiByte()

Syntax

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

Purpose

This routine converts an entire `NULL`-terminated wide character string into a multibyte string. The output buffer is `NULL`-terminated. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE` or `OCI_ERROR`

Parameters

hndl(IN/OUT)
OCI environment or user session handle to determine the character set of string

dst(OUT)
Destination buffer for multibyte string

src(IN)
Source `wchar` string to be converted

srcsz(IN)
Length of source string in characters

rsize(OUT)
Number of bytes written into destination buffer. If it is a `NULL` pointer, then nothing is returned.

OCIWideCharInSizeToMultiByte()

Syntax

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

Purpose

This routine converts part of `wchar` string into the multibyte format. It converts as many complete characters as it can until it reaches the output buffer size or the input buffer size or until it reaches a `NULL` terminator in source string. The output buffer is `NULL`-terminated if space permits. If `dstsz` is zero, the function just returns the size of byte not including the `NULL` terminator needed to store the converted string. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE` or `OCI_ERROR`

Parameters

hndl(IN/OUT)
OCI environment or user session handle to determine the character set of string

dst(OUT)
Destination buffer for multibyte. It can be a `NULL` pointer if `dstsz` is zero

dstsz(IN)
Destination buffer size in bytes. If it is zero, it returns the size in bytes need for converted string.

src(IN)
Source `wchar` string to be converted

srcsz(IN)
Length of source string in characters

rsize(OUT)
Number of bytes written into destination buffer, or number of bytes need to store the converted string if `dstsz` is zero. If it is a `NULL` pointer, nothing is returned.

OCIWideCharToLower()

Syntax

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

Purpose

This function converts the `wchar` character specified by `wc` into the corresponding lowercase character if it exists in the specified locale. If no corresponding lowercase character exists, then it returns `wc` itself. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

A `wchar`

Parameters

hndl(IN/OUT)

OCI environment or user session handle to determine the character set

wc(IN)

`wchar` for lowercase conversion

OCIWideCharToUpper()

Syntax

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

Purpose

This function converts the `wchar` character specified by `wc` into the corresponding uppercase character if it exists in the specified locale. If no corresponding uppercase character exists, then it returns `wc` itself. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

A `wchar`

Parameters

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

wc(IN)
`wchar` for uppercase conversion

OCIWideCharStrcmp()

Syntax

```
int OCIWideCharStrcmp(dvoid *hndl, CONST OCIWchar *wstr1, CONST OCIWchar *wstr2,
int flag)
```

Purpose

It compares two `wchar` strings by binary (based on `wchar` encoding value), linguistic, or case-insensitive comparison methods. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

- 0, if `wstr1 = wstr2`
- Positive, if `wstr1 > wstr2`
- Negative, if `wstr1 < wstr2`

Parameters

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

wstr1(IN)
Pointer to a `NULL`-terminated `wchar` string

wstr2(IN)
Pointer to a `NULL`-terminated `wchar` string

flag(IN)
Used to decide the comparison method. It can take one of the following values:

- `OCI_NLS_BINARY`: Binary comparison. This is the default value.
- `OCI_NLS_LINGUISTIC`: Linguistic comparison specified in the locale definition.

This flag can be used with `OCI_NLS_CASE_INSENSITIVE` for case-insensitive comparison. For example, use `OCI_NLS_LINGUISTIC|OCI_NLS_CASE_INSENSITIVE` to compare strings linguistically without regard to case.

OCIWideCharStrncmp()

Syntax

```
int OCIWideCharStrncmp(dvoid *hndl, CONST OCIWchar *wstr1, size_t len1, CONST
OCIWchar *wstr2, size_t len2, int flag)
```

Purpose

This function is similar to `OCIWideCharStrcmp()`. It compares two wide character strings by binary, linguistic, or case-insensitive comparison methods. At most `len1` bytes from `wstr1` and `len2` bytes from `wstr2` are compared. The `NULL` terminator is used in the comparison. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

- 0, if `wstr1 = wstr2`
- Positive, if `wstr1 > wstr2`
- Negative, if `wstr1 < wstr2`

Parameters

hndl(IN/OUT)

OCI environment or user session handle to determine the character set

wstr1(IN)

Pointer to the first `wchar` string

len1(IN)

The length for the first string for comparison

wstr2(IN)
Pointer to the second `wchar` string

len2(IN)
The length for the second string for comparison

flag(IN)
It is used to decide the comparison method. It can take one of the following values:

- `OCI_NLS_BINARY`: For the binary comparison, this is default value.
- `OCI_NLS_LINGUISTIC`: For the linguistic comparison specified in the locale.

This flag can be used with `OCI_NLS_CASE_INSENSITIVE` for case-insensitive comparison. For example, use `OCI_NLS_LINGUISTIC|OCI_NLS_CASE_INSENSITIVE` to compare strings linguistically without regard to case.

OCIWideCharStrcat()

Syntax

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

Purpose

This function appends a copy of the `wchar` string pointed to by `wsrcstr`, including the `NULL` terminator to the `wchar` string pointed to by `wdststr`. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

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

Parameters

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

OCIWideCharStrncat()

Syntax

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

Purpose

This function is similar to `OCIWideCharStrcat()`. At most *n* characters from `wsrcstr` are appended to `wdststr`. Note that the `NULL` terminator in `wsrcstr` stops appending. `wdststr` is `NULL`-terminated. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of characters in the result string, not including the `NULL` terminator

Parameters

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

wdststr(IN/OUT)
Pointer to the destination `wchar` string to append

wsrcstr(IN)
Pointer to the source `wchar` string to append

n(IN)
Number of characters from `wsrcstr` to append

OCIWideCharStrchr()

Syntax

```
OCIWchar *OCIWideCharStrchr(dvoid *hndl, CONST OCIWchar *wstr, OCIWchar wc)
```

Purpose

This function searches for the first occurrence of `wc` in the `wchar` string pointed to by `wstr`. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

A `wchar` pointer if successful, otherwise a `NULL` pointer

Parameters

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

wstr(IN)
Pointer to the `wchar` string to search

wc(IN)
`wchar` to search for

OCIWideCharStrrchr()

Syntax

```
OCIWchar *OCIWideCharStrrchr(dvoid *hndl, CONST OCIWchar *wstr, OCIWchar wc)
```

Purpose

This function searches for the last occurrence of `wc` in the `wchar` string pointed to by `wstr`. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

`wchar` pointer if successful, otherwise a `NULL` pointer

Parameters

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

wstr(IN)
Pointer to the `wchar` string to search

wc(IN)
`wchar` to search for

OCIWideCharStrcpy()

Syntax

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

Purpose

This function copies the `wchar` string pointed to by `wsrcstr`, including the `NULL` terminator, into the array pointed to by `wdststr`. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of characters copied not including the `NULL` terminator

Parameters

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

wdststr(OUT)
Pointer to the destination `wchar` buffer

wsrcstr(IN)
Pointer to the source `wchar` string

OCIWideCharStrncpy()

Syntax

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

Purpose

This function is similar to OCIWideCharStrcpy(), except that at most *n* characters are copied from the array pointed to by wsrcstr to the array pointed to by wdststr. Note that the NULL terminator in wdststr stops copying and the result string is NULL-terminated. If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate function, then this function produces an error.

Returns

The number of characters copied not including the NULL terminator

Parameters

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

wdststr(OUT)
Pointer to the destination wchar buffer

wsrcstr(IN)
Pointer to the source wchar string

n(IN)
Number of characters from wsrcstr to copy

OCIWideCharStrlen()

Syntax

```
size_t OCIWideCharStrlen(dvoid *hndl, CONST OCIWchar *wstr)
```

Purpose

This function computes the number of characters in the wchar string pointed to by wstr, not including the NULL terminator, and returns this number. If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate function, then this function produces an error.

Returns

The number of characters not including the NULL terminator

Parameters

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

wstr(IN)
Pointer to the source `wchar` string

OCIWideCharStrCaseConversion()

Syntax

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

Purpose

This function converts the wide `char` string pointed to by `wsrcstr` into the upper case or lower case specified by the flag and copies the result into the array pointed to by `wdststr`. The result string is `NULL`-terminated. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of characters for the result string, not including the `NULL` terminator

Parameters

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 upper case

- OCI_NLS_LOWERCASE: Convert to lower case

This flag can be used with OCI_NLS_LINGUISTIC to specify that the linguistic setting in the locale is used for case conversion.

OCIWideCharDisplayLength()

Syntax

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

Purpose

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. If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate function, then this function produces an error.

Returns

The number of display positions

Parameters

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)
```

Purpose

This function determines the number of bytes required for wc in multibyte encoding. If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate function, then this function produces an error.

Returns

The number of bytes required for `wc`

Parameters

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)
```

Purpose

It compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

- 0, if `str1 = str2`
- Positive, if `str1 > str2`
- Negative, if `str1 < str2`

Parameters

hndl(IN/OUT)

OCI environment or user session handle

str1(IN)

Pointer to a `NULL`-terminated string

str2(IN)

Pointer to a `NULL`-terminated string

flag(IN)
It is used to decide the comparison method. It can take one of the following values:

- `OCI_NLS_BINARY`: Binary comparison This is the default value.
- `OCI_NLS_LINGUISTIC`: Linguistic comparison specified in the locale

This flag can be used with `OCI_NLS_CASE_INSENSITIVE` for case-insensitive comparison. For example, use `OCI_NLS_LINGUISTIC|OCI_NLS_CASE_INSENSITIVE` to compare strings linguistically without regard to case.

OCIMultiByteStrncmp()

Syntax

```
int OCIMultiByteStrncmp(dvoid *hndl, CONST OraText *str1, size_t len1, OraText
*str2, size_t len2, int flag)
```

Purpose

This function is similar to `OCIMultiByteStrcmp`(), except that at most `len1` bytes from `str1` and `len2` bytes from `str2` are compared. The `NULL` terminator is used in the comparison. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

- 0, if `str1` = `str2`
- Positive, if `str1` > `str2`
- Negative, if `str1` < `str2`

Parameters

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`: Binary comparison. This is the default value.
- `OCI_NLS_LINGUISTIC`: Linguistic comparison specified in the locale

This flag can be used with `OCI_NLS_CASE_INSENSITIVE` for case-insensitive comparison. For example, use `OCI_NLS_LINGUISTIC|OCI_NLS_CASE_INSENSITIVE` to compare strings linguistically without regard to case.

OCIMultiByteStrcat()

Syntax

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

Purpose

This function appends a copy of the multibyte string pointed to by `srcstr`, including the `NULL` terminator to the end of string pointed to by `dststr`. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of bytes in the result string, not including the `NULL` terminator

Parameters

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

dststr(IN/OUT)
Pointer to the destination multibyte string for appending

srcstr(IN)
Pointer to the source string to append

OCIMultiByteStrncat()

Syntax

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

Purpose

This function is similar to `OCIMultiByteStrcat`(). At most *n* bytes from `srcstr` are appended to `dststr`. Note that the `NULL` terminator in `srcstr` stops appending and the function appends as many character as possible within *n* bytes. `dststr` is `NULL`-terminated. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of bytes in the result string, not including the `NULL` terminator

Parameters

hndl(IN/OUT)
Pointer to OCI environment or user session handle

dststr(IN/OUT)
Pointer to the destination multibyte string for appending

srcstr(IN)
Pointer to the source multibyte string to append

n(IN)
The number of bytes from `srcstr` to append

OCIMultiByteStrcpy()

Syntax

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

Purpose

This function copies the multibyte string pointed to by `srcstr`, including the `NULL` terminator, into the array pointed to by `dststr`. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of bytes copied, not including the `NULL` terminator

Parameters

hndl(IN/OUT)
Pointer to the OCI environment or user session handle

dststr(OUT)
Pointer to the destination buffer

srcstr(IN)
Pointer to the source multibyte string

OCIMultiByteStrncpy()

Syntax

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

Purpose

This function is similar to `OCIMultiByteStrcpy()`. At most *n* bytes are copied from the array pointed to by `srcstr` to the array pointed to by `dststr`. Note that the `NULL` terminator in `srcstr` stops copying and the function copies as many characters as possible within *n* bytes. The result string is `NULL`-terminated. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of bytes copied not including the `NULL` terminator

Parameters

hndl(IN/OUT)
Pointer to OCI environment or user session handle

srcstr(OUT)
Pointer to the destination buffer

dststr(IN)
Pointer to the source multibyte string

n(IN)
The number of bytes from `srcstr` to copy

OCIMultiByteStrlen()

Syntax

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

Purpose

This function returns the number of bytes in the multibyte string pointed to by `str`, not including the `NULL` terminator. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of bytes not including the `NULL` terminator

Parameters

hndl(IN/OUT)
Pointer to the OCI environment or user session handle

str(IN)
Pointer to the source multibyte string

OCIMultiByteStrnDisplayLength()

Syntax

```
size_t OCIMultiByteStrnDisplayLength(dvoid *hndl, CONST OraText *str1, size_t n)
```

Purpose

This function returns the number of display positions occupied by the complete characters within the range of *n* bytes. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of display positions

Parameters

hndl(IN/OUT)
OCI environment or user session handle

str(IN)
Pointer to a multibyte string

n(IN)
The number of bytes to examine

OCIMultiByteStrCaseConversion()

Syntax

```
size_t OCIMultiByteStrCaseConversion(dvoid *hndl, OraText *dststr, CONST OraText
*srcstr, ub4 flag)
```

Purpose

This function converts the multibyte string pointed to by `srcstr` into upper case or lower case as specified by the flag and copies the result into the array pointed to by `dststr`. The result string is `NULL`-terminated. If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

The number of bytes for result string, not including the NULL terminator

Parameters

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 which to convert:

- OCI_NLS_UPPERCASE: Convert to upper case
- OCI_NLS_LOWERCASE: Convert to lower case

This flag can be used with OCI_NLS_LINGUISTIC to specify that the linguistic setting in the locale is used for case conversion.

Example: Manipulating Strings in OCI

The following example shows a simple case of manipulating strings.

```
size_t MyConvertMultiByteToWideChar(envhp, dstBuf, dstSize, srcStr)
OCIEnv     *envhp;
OCIWchar   *dstBuf;
size_t      dstSize;
OraText       *srcStr;                         /* null terminated source string
*/
{
  sword  ret;
  size_t dstLen = 0;
  size_t srcLen;

  /* get length of source string */
  srcLen = OCIMultiByteStrlen(envhp, srcStr);

  ret = OCIMultiByteInSizeToWideChar(envhp,       /* environment handle */
                 dstBuf,                              /* destination buffer */
```

```
                   dstSize,                       /* destination buffer size */
                   srcStr,                                  /* source string */
                   srcLen,                        /* length of source string */
                   &dstLen);                /* pointer to destination length */

  if (ret != OCI_SUCCESS)
  {
    checkerr(envhp, ret, OCI_HTYPE_ENV);
  }
  return(dstLen);
}
```

See Also: *Oracle Call Interface Programmer's Guide*

Classifying Characters in OCI

Table 8–2 shows the OCI character classification functions. They are described in more detail in the rest of this section.

Table 8–2 OCI Character Classification Functions

Function	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 multibyte

OCIWideCharIsAlnum()

Syntax

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

Purpose

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

Returns

`TRUE` or `FALSE`

Parameters

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)
```

Purpose

It tests whether `wc` is an alphabetic letter.

Returns

`TRUE` or `FALSE`

Parameters

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)
```

Purpose

It tests whether `wc` is a control character.

Returns

`TRUE` or `FALSE`

Parameters

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)
```

Purpose

It tests whether `wc` is a decimal digit character.

Returns

`TRUE` or `FALSE`

Parameters

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)
```

Purpose
It tests whether `wc` is a graph character. A graph character is a character with a visible representation and normally includes alphabetic letters, decimal digits, and punctuation.

Returns
TRUE or FALSE

Parameters

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)
```

Purpose
It tests whether `wc` is a lowercase letter.

Returns
TRUE or FALSE

Parameters

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)
```

Purpose

It tests whether `wc` is a printable character.

Returns

TRUE or FALSE

Parameters

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

wc(IN)
`wchar` for testing

OCIWideCharIsPunct()

Syntax

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

Purpose

It tests whether `wc` is a punctuation character.

Returns

TRUE or FALSE

Parameters

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)
```

Purpose

It tests whether `wc` is a space character. A space character causes white space only in displayed text (for example, space, tab, carriage return, new line, vertical tab or form feed).

Returns

TRUE or FALSE

Parameters

hndl(IN/OUT)

OCI environment or user session handle to determine the character set

wc(IN)

`wchar` for testing

OCIWideCharIsUpper()

Syntax

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

Purpose

It tests whether `wc` is an uppercase letter.

Returns

`TRUE` or `FALSE`

Parameters

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)
```

Purpose

It tests whether `wc` is a hexadecimal digit (0-9, A-F, a-f).

Returns

`TRUE` or `FALSE`

Parameters

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)`

Purpose

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

Returns

`TRUE` or `FALSE`

Parameters

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

wc(IN)
`wchar` for testing

Example: Classifying Characters in OCI

The following example shows how to classify characters in OCI.

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

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


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

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

  return(status);
}
```

Converting Character Sets in OCI

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

Table 8–3 summarizes the OCI character set conversion functions. They are described in more detail in the rest of this section.

Table 8–3 OCI Character Set Conversion Functions

Function	Description
`OCICharsetToUnicode()`	Converts a multibyte string pointed to by `src` to Unicode into the array pointed to by `dst`
`OCIUnicodeToCharset()`	Converts a Unicode string pointed to by `src` to multibyte into the array pointed to by `dst`
`OCINlsCharSetConvert()`	Converts a string from one character set to another
`OCICharSetConversionIsReplacementUsed()`	Indicates whether replacement characters were used for characters that could not be converted in the last invocation of `OCINlsCharsetConvert()` or `OCICharSetToUnicode()`

OCICharSetToUnicode()

Syntax

```
sword OCICharSetToUnicode(dvoid *hndl, ub2 *dst, size_t dstlen, CONST OraText
*src, size_t srclen, size_t *rsize)
```

Purpose

This function converts a multibyte string pointed to by `src` to Unicode into the array pointed to by `dst`. The conversion stops when it reaches the source limitation or destination limitation. The function returns the number of characters converted into a Unicode string. If `dstlen` is `0`, then the function scans the string, counts the number of characters, and returns the number of characters into `rsize`, but does not convert the string.

If OCI_UTF16ID is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE` or `OCI_ERROR`

Parameters

hndl(IN/OUT)
Pointer to an OCI environment or user session handle

dst(OUT)
Pointer to a destination buffer

dstlen(IN)
The size of the destination buffer in characters

src(IN)
Pointer to a multibyte source string

srclen(IN)
The size of the source string in bytes

rsize(OUT)
The number of characters converted. If it is a `NULL` pointer, then nothing is returned.

OCIUnicodeToCharSet()

Syntax

```
sword OCIUnicodeToCharSet(dvoid *hndl, OraText *dst, size_t dstlen, CONST ub2
*src, size_t srclen, size_t *rsize)
```

Purpose

This function converts a Unicode string pointed to by `src` to a multibyte string into the array pointed to by `dst`. The conversion stops when it reaches the source limitation or destination limitation. The function returns the number of bytes converted into a multibyte string. If `dstlen` is zero, it returns the number of bytes into `rsize` without conversion.

If a Unicode character is not convertible for the character set specified in OCI environment or user session handle, a replacement character is used for it. In this case, `OCICharsetConversionIsReplacementUsed()` returns `TRUE`.

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate` function, then this function produces an error.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE` or `OCI_ERROR`

Parameters

hndl(IN/OUT)
Pointer to an OCI environment or user session handle

dst(OUT)
Pointer to a destination buffer

dstlen(IN)
The size of destination buffer in bytes

src(IN)
Pointer to a Unicode string

srclen(IN)
The size of the source string in characters

rsize(OUT)
The number of bytes converted. If it is a `NULL` pointer, nothing is returned.

OCINlsCharSetConvert()

Syntax

```
sword OCINlsCharSetConvert(dvoid *envhp, OCIError *errhp,ub2 dstid, dvoid *dstp,
size_t dstlen,ub2 srcid, CONST dvoid *srcp, size_tsrclen, size_t *rsize);
```

Purpose

This function converts a string pointed to by `src` in the character set specified by `srcid` to the array pointed to by `dst` in the character set specified by `dstid`. The conversion stops when it reaches the data size limitation of either the source or the destination. The function returns the number of bytes converted into the destination buffer. Although either the source or the destination character set ID can be specified as `OCI_UTF16ID`, the length of the original and the converted data is represented in bytes, rather than number of characters. Note that the conversion does not stop when it encounters null data. To get the character set ID from the character set name, use `OCINlsCharSetNameToId()`. To check if derived data in the destination buffer contains replacement characters, use `OCICharSetConversionIsReplacementUsed()`. The buffers should be aligned with the byte boundaries appropriate for the character sets. For example, the `ub2` datatype should be used to hold strings in UTF-16.

Returns

`OCI_SUCCESS` or `OCI_ERROR`; number of bytes converted

Parameters

errhp(IN/OUT)

OCI error handle. If there is an error, it is recorded in `errhp` and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.

dstid(IN)

Character set ID for the destination buffer

dstp(OUT)

Pointer to the destination buffer

dstlen(IN)

The maximum size in bytes of the destination buffer

srcid(IN)

Character set ID for the source buffer

srcp(IN)

Pointer to the source buffer

srclen(IN)
The length in bytes of the source buffer

rsize(OUT)
The number of characters converted. If the pointer is `NULL`, then nothing is returned.

OCICharSetConversionIsReplacementUsed()

Syntax
`boolean OCICharSetConversionIsReplacementUsed(dvoid *hndl)`

Purpose
This function indicates whether the replacement character was used for characters that could not be converted during the last invocation of `OCICharSetToUnicode()` or `OCICharSetConvert()`.

Returns
The function returns `TRUE` if the replacement character was used when `OCICharSetConvert()` or `OCICharSetToUnicode()` was last invoked. Otherwise the function returns `FALSE`.

Parameter

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 are used if there is no mapping for a character from Unicode to the Oracle character set. Thus, not every character can make a round-trip conversion to the original character. Data loss occurs with certain characters.

Example: Converting Character Sets in OCI
The following example shows 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);
}
```

OCI Messaging Functions

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

See Also: *Oracle9i Data Cartridge Developer's Guide*

Table 8–4 summarizes the OCI messaging functions.

Table 8–4 OCI Messaging Functions

Function	Description
`OCIMessageOpen()`	Opens a message handle in a language pointed to by `hndl`
`OCIMessageGet()`	Retrieves a message with message number identified by `msgno`. If the buffer is not zero, then the function copies 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

This section contains the following topics:

- OCIMessageOpen()
- OCIMessageGet()
- OCIMessageClose()
- Example: Retrieving a Message from a Text Message File
- lmsgen Utility

OCIMessageOpen()

Syntax

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

Purpose

This function opens a message-handling facility in a language pointed to by `hndl`. It first tries to open the message file corresponding to `hndl`. If it succeeds, then it uses that file to initialize a message handle. If it cannot find the message file that corresponds to the language, it looks for a primary language file as a fallback. For example, if the Latin American Spanish file is not found, then it tries to open the Spanish file. If the fallback fails, then it uses the default message file, whose language is `AMERICAN`. The function returns a pointer to a message handle into the `msghp` parameter.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`

Parameters

hndl(IN/OUT)
Pointer to an OCI environment or user session handle for message language

errhp(IN/OUT)
The OCI error handle. If there is an error, it is recorded in `errhp,` and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.

msghp(OUT)
A message handle for return

product(IN)
A pointer to a product name. The product name is used to locate the directory for messages. Its location depends on the operating system. For example, in Solaris, the directory of message files for the `rdbms` product is `$ORACLE_HOME/rdbms`.

facility(IN)
A pointer to a facility name in the product. It is used to construct a message file name. A message file name follows the conversion with `facility` as prefix. For example, the message file name for the `img` facility in the American language is `imgus.msb`, where `us` is the abbreviation for the American language and `msb` is the message binary file extension.

dur(IN)
The duration for memory allocation for the return message handle. It can have the following values:

```
OCI_DURATION_PROCESS
OCI_DURATION_SESSION
OCI_DURATION_STATEMENT
```

OCIMessageGet()

Syntax

```
OraText *OCIMessageGet(OCIMsg *msgh, ub4 msgno, OraText *msgbuf, size_t buflen)
```

Purpose
This function gets a message with the message number identified by `msgno`. If `buflen` is not zero, then the function copies the message into the buffer pointed to

by `msgbuf`. If `buflen` is zero, then the message is copied into a message buffer inside the message handle pointed to by `msgh`.

Returns

It returns the pointer to the `NULL`-terminated message string. If the translated message cannot be found, then it tries to return the equivalent English message. If the equivalent English message cannot be found, then it returns a `NULL` pointer.

Parameters

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 for the retrieved message. If `buflen` is zero, then it can be a `NULL` pointer.

buflen(IN)
The size of the destination buffer

OCIMessageClose()

Syntax

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

Purpose

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`

Parameters

Table 8–5 OCIMessageClose Keywords/Parameters (Cont.)

Keyword/Parameter	Meaning
`hndl(IN/OUT)`	Pointer to an OCI environment or user session handle for message language
`errhp(IN/OUT)`	The OCI error handle. If there is an error, it is recorded in `errhp` and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.
`msgh(IN/OUT)`	A pointer to a message handle that was previously opened by `OCIMessageOpen()`

Example: Retrieving a Message from a Text Message File

This example creates a message handle, initializes it to retrieve messages from `impus.msg`, retrieves message number 128, and closes the message handle. It assumes that OCI environment handles, OCI session handles, product, facility, and cache size have been 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 there are no more messages to retrieve */
OCIMessageClose(hndl, errhp, msghnd);
```

lmsgen Utility

Purpose

The `lmsgen` utility converts text-based message files (`.msg`) into binary format (`.msb`) so that Oracle messages and OCI messages provided by the user can be returned to OCI functions in the desired language.

Syntax

`LMSGEN` *`text_file product facility`* `[`*`language`*`]`

text_file is a message text file.
product is the name of the product.
facility is the name of the facility.

language is the optional message language corresponding to the language specified in the NLS_LANG parameter. The language parameter is required if the message file is not tagged properly with language.

Text Message Files

Text message files must follow these guidelines:

- Lines that start with / and // are treated as internal comments and are ignored.
- To tag the message file with a specific language, include a line similar to the following:

  ```
  #   CHARACTER_SET_NAME= Japanese_Japan.JA16EUC
  ```

- Each message contains 3 fields:

  message_number, *warning_level*, *message_text*

  The message number must be unique within a message file.
  The warning level is not currently used. Use 0.
  The message text cannot be longer than 76 bytes.

The following is an example of an Oracle message text file:

```
/ Copyright (c) 2001 by the Oracle Corporation.  All rights reserved.
/ This is a test us7ascii message file
# CHARACTER_SET_NAME= american_america.us7ascii
/
00000, 00000, "Export terminated unsuccessfully\n"
00003, 00000, "no storage definition found for segment(%lu, %lu)"
```

Example: Creating a Binary Message File from a Text Message File

The following table contains sample values for the lmsgen parameters:

Parameter	Value
product	$HOME/myApplication
facility	imp

Parameter	Value
language	AMERICAN
text_file	impus.msg

The text message file is found in the following location:

```
$HOME/myApp/mesg/impus.msg
```

One of the lines in the text message file is:

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

The `lmsgen` utility converts the text message file (`impus.msg`) into binary format, resulting in a file called `impus.msb`:

```
% lmsgen impus.msg $HOME/myApplication imp AMERICAN
```

The following output results:

```
Generating message file impus.msg -->
/home/scott/myApplication/mesg/impus.msb

NLS Binary Message File Generation Utility: Version 9.2.0.0.0 -Production

Copyright (c) Oracle Corporation 1979, 2001.  All rights reserved.

CORE    9.2.0.0.0       Production
```

9 Java Programming in a Global Environment

This chapter examines globalization support for individual Java components. It includes the following topics:

Overview of Oracle9*i* Java Support

Java support is included in all tiers of a multitier computing environment so that you can develop and deploy Java programs. You can run Java classes as Java stored procedures on the Java Virtual Machine (Oracle JVM) of the Oracle9*i* database. You can develop a Java class, load it into the database, and package it as a stored procedure that can be called from SQL.

The JDBC driver and SQLJ translator are also provided as programmatic interfaces that enable Java programs to access the Oracle9*i* database. You can write a Java application using JDBC or SQLJ programs with embedded SQL statements to access the database. Globalization support is provided across these Java components to ensure that they function properly across databases with different character sets and language environments, and that they enable the development and deployment of multilingual Java applications for Oracle9*i*.

This chapter examines globalization support for individual Java components. Typical database and client configurations for multilingual application deployment are discussed, including an explanation of how the Java components are used in the configurations. The design and implementation of a sample application are used to demonstrate how Oracle's Java support makes the application run in a multilingual environment.

Java components provide globalization support and use Unicode as the multilingual character set. Table 9–1 shows the Java components of Oracle9*i*.

Table 9–1 Oracle9i Java Components

Java Component	Description
JDBC driver	Oracle provides JDBC as the core programmatic interface for accessing Oracle9*i* databases. There are four JDBC drivers provided by Oracle: two for client access and two for server access. ■ The JDBC OCI driver is used by Java applications. ■ The JDBC thin driver is primarily used by Java applets. ■ The Oracle JDBC server-side thin driver offers the same functionality as the client-side JDBC thin driver and is used primarily by Java classes running on the Java VM of the database server to access a remote database. The JDBC server-side internal driver is a server-side driver that is used by Java classes running on the Java VM of the database server.

Table 9–1 Oracle9i Java Components (Cont.)

Java Component	Description
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 Virtual Machine (JVM)	A Java VM based on the JDK is integrated into the database server that enables the running of Java classes as Java stored procedures. It comes with a set of supporting services such as the library manager, which manages Java classes stored in the database.

Globalization Support for JDBC Drivers

Oracle JDBC drivers provide globalization support by allowing you to retrieve data from or insert data into columns of the SQL `CHAR` and `NCHAR` datatypes of an Oracle9*i* database. Because Java strings are encoded as UTF-16 (16-bit Unicode) for JDBC programs, the target character set on the client is always UTF-16. For data stored in the `CHAR`, `VARCHAR2`, `LONG`, and `CLOB` datatypes, JDBC transparently converts the data from the database character set to UTF-16. For Unicode data stored in the `NCHAR`, `NVARCHAR2`, and `NCLOB` datatypes, JDBC transparently converts the data from the national character set to UTF-16.

The following examples are commonly used Java methods for JDBC that rely heavily on character set conversion:

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

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.

This section includes the following topics:

- Accessing SQL CHAR Datatypes Using JDBC
- Accessing SQL NCHAR Datatypes Using JDBC
- Using the oracle.sql.CHAR Class
- Restrictions on Accessing SQL CHAR Data with JDBC

Accessing SQL CHAR Datatypes Using JDBC

To insert a Java string into a database column of a SQL `CHAR` datatype, you can use the `PreparedStatement.setString()` method to specify the bind variable. Oracle's JDBC drivers transparently convert the Java string to the database character set. The following example shows how to bind a Java string `last_name` to a `VARCHAR2` column `last_name`.

```
int employee_id= 12345;
String last_name= "\uFF2A\uFF4F\uFF45";
PreparedStatement pstmt =
    conn.prepareStatement ("INSERT INTO employees (employee_id, last_name)
    VALUES(?,?)");
pstmt.setInt(1, employee_id);
pstmt.setString(2, last_name);
pstmt.execute();
pstmt.close();
```

For data stored in SQL `CHAR` datatypes, the techniques that Oracle's drivers use to perform character set conversion for Java applications depend on the character set that the database uses. The simplest case is when the database uses a US7ASCII or WE8ISO8859P1 character set. In this case, the driver converts the data directly from the database character set to UTF-16,which is used in Java applications.

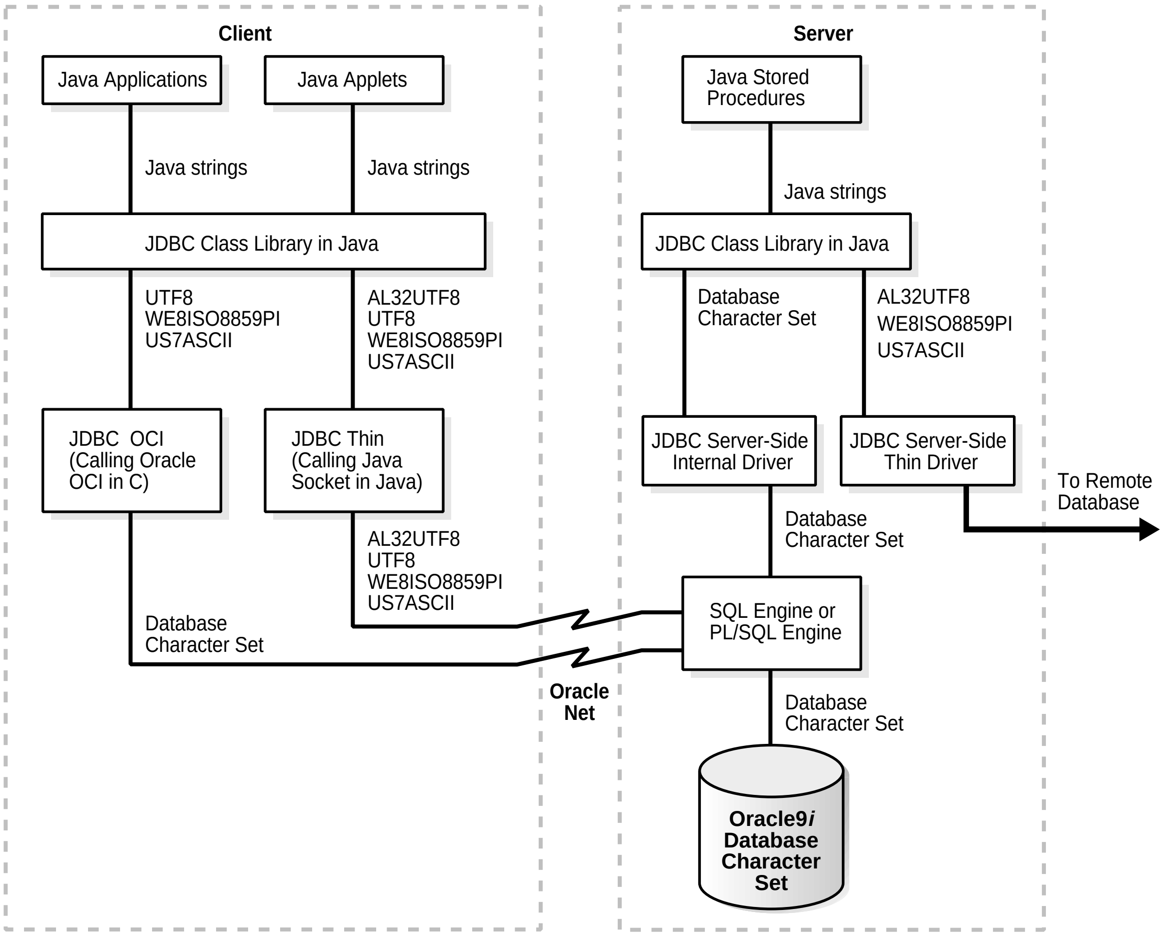
If you are working with databases that employ a character set that is not US7ASCII or WE8ISO8859P1 (for example, JA16SJIS or KO16KSC5601), then the driver converts the data first to UTF-8, then to UTF-16. The following sections describe the conversion paths for different JDBC drivers:

- JDBC Class Library Character Set Conversion
- JDBC OCI Driver Character Set Conversion
- JDBC Thin Driver Character Set Conversion

- JDBC Server-Side Internal Driver Character Set Conversion

Figure 9–1 shows how data is converted in JDBC drivers.

Figure 9–1 JDBC Data Conversion



JDBC Class Library Character Set Conversion

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 AL32UTF8 data for the JDBC thin driver and database character set data for the JDBC server-side driver. The JDBC Class Library converts the input stream to UTF-16 before passing it to the client applications. AL32UTF8 is another character set in addition to UTF8 for encoding Unicode characters in the UTF-8 encoding. It supports supplemental Unicode characters. If the input stream is in UTF8 or AL32UTF8, then the JDBC Class Library converts the UTF8 or AL32UTF8 encoded string to UTF-16 by using the bit-wise operation defined in the UTF-8 to UTF-16 conversion algorithm. If the input stream is in US7ASCII or WE8ISO8859P1, then it converts the input string to UTF-16 by casting the bytes to Java characters. If the input stream is not one of US7ASCII, WE8ISO8859P1, UTF8 and AL32UTF8, then the JDBC Class Library converts the input stream by calling the Oracle character set conversion facility. This conversion path is only used for the JDBC server-side driver.

JDBC OCI Driver Character Set Conversion

In the case of a JDBC OCI driver, there is a client-side character set as well as a database character set. The client character set is determined at client start time by the value of the `NLS_LANG` environment variable on the client. 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. When performing character set conversion, the JDBC OCI driver has to take three factors into consideration:

- The database character set and language
- The client character set and language
- The Java application's character set

The JDBC OCI driver transfers the data from the server to the client in the character set of the database. Depending on the value of the `NLS_LANG` environment variable, the driver handles character set conversions in one of two ways:

- If the value of `NLS_LANG` is not specified, or if it is set to the US7ASCII or WE8ISO8859P1 character set, then the JDBC OCI driver uses Java to convert the character set from US7ASCII or WE8ISO8859P1 directly to UTF-16 in the JDBC Class Library.
- If the value of `NLS_LANG` is set to a character set other than US7ASCII or WE8ISO8859P1, then the driver uses UTF8 as the client character set. This happens automatically and does not require any user intervention. OCI then converts the data from the database character set to UTF8. The JDBC OCI driver then passes the UTF8 data to the JDBC Class Library where the UTF8 data is converted to UTF-16.

JDBC Thin Driver Character Set Conversion

If 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 of the JDBC thin driver are different from conversion routines of the JDBC OCI driver.

If the database character set is US7ASCII, WE8ISO8859P1, UTF8, or AL32UTF8, then the data is transferred to the client without any conversion. The JDBC Class Library then converts the data to UTF-16 in Java.

Otherwise, the server first translates the data to UTF8 or AL32UTF8 before transferring it to the client. On the client, the JDBC Class Library converts the data to UTF-16 in Java.

JDBC Server-Side Internal Driver Character Set Conversion

For Java classes running in the Java VM of the Oracle9*i* Server, the JDBC server-side internal driver is used to talk to the SQL engine or the PL/SQL engine for SQL processing. Because the JDBC server-side internal driver is running in the same address space as the Oracle server process, it makes a local function call to the SQL engine or the PL/SQL engine. Data sent to or returned from the SQL engine or the PL/SQL engine is encoded in the database character set, No data conversion is performed in the JDBC server-side internal driver, and the data is passed to or from the JDBC Class Library as is. Any necessary conversion is delegated to the JDBC Class Library.

Accessing SQL NCHAR Datatypes Using JDBC

JDBC enables Java programs to access columns of the SQL `NCHAR` datatypes in an Oracle9*i* database. Data conversion for the SQL `NCHAR` datatypes is different from data conversion for the SQL `CHAR` datatypes. All Oracle JDBC drivers convert data in the SQL `NCHAR` column from the national character set, which is either UTF8 or AL16UTF16, directly to UTF-16 encoded Java strings. In the following Java program, you can bind a Java string `last_name` to an `NVARCHAR2` column `last_name`:

```
int employee_id = 12345;
String ename = "\uFF2A\uFF4F\uFF45";
oracle.jdbc.OraclePreparedStatement pstmt =
    (oracle.jdbc.OraclePreparedStatement)
    conn.prepareStatement("INSERT INTO employees (empoyee_id, last_name) VALUES
(?, ?)");
pstmt.setFormOfUse(2, oracle.jdbc.OraclePreparedStatement.FORM_NCHAR);
pstmt.setInt(1, employee_id);
pstmt.setString(2, last_name);
pstmt.execute();
pstmt.close();
```

See Also: "Binding and Defining Java Strings in Unicode" on page 6-26 for more information about programming against the SQL NCHAR datatypes

Using the oracle.sql.CHAR Class

The `oracle.sql.CHAR` class has a special functionality for conversion of character data. The Oracle character set is a key attribute of the `oracle.sql.CHAR` class. The Oracle character set is always passed in when an `oracle.sql.CHAR` object is constructed. Without a known character set, the bytes of data in the `oracle.sql.CHAR` object are meaningless.

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, then `getString()` returns a `SQLException`.

- `toString()`

  Identical to `getString()`, except that if the character set is not recognized, then `toString()` returns a hexadecimal representation of the `oracle.sql.CHAR` data and does not returns a `SQLException`.

- `getStringWithReplacement()`

  Identical to `getString()`, except that 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 among character sets, but it is often a question mark.

You may 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 using an instance of the `oracle.sql.CharacterSet` class. Each instance of the `oracle.sql.CharacterSet` class represents one of the character sets that Oracle supports.

Complete the following tasks to construct an `oracle.sql.CHAR` object:

1. Create a `CharacterSet` instance by calling the static `CharacterSet.make()` method. This method creates the character set class. It requires as input a valid Oracle character set (`OracleId`). 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 specified as *`Oracle_character_set_name`*`_CHARSET` where *`Oracle_character_set_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.

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 between the server character set and the client character set.

Inserting and Retrieving Data with the oracle.sql.CHAR Class

When you call the `OracleResultSet.getCHAR()` method to get a bind variable as an `oracle.sql.CHAR` object, JDBC constructs and populates the `oracle.sql.CHAR` objects after character data has been read from the database. Similarly, you can call the `OraclePreparedStatement.sql.CHAR()` method to set a bind variable using an `oracle.sql.CHAR` object. For example:

```
int employee_id = 12345;
String ename = "\uFF2A\uFF4F\uFF45";
String eaddress = "Address of \uFF2A\uFF4F\uFF45";
/* CharacterSet object for VARCHAR2 column */
CharacterSet dbCharset = CharacterSet.make(CharacterSet.JA16SJIS_CHARSET);
```

```
/* CharacterSet object for NVARCHAR2 column */
CharacterSet ncCharset = CharacterSet.make(CharacterSet.AL16UTF16_CHARSET);

/* last_name is in VARCHAR2 and address is in NVARCHAR2 */
oracle.jdbc.OraclePreparedStatement pstmt =
    (oracle.jdbc.OraclePreparedStatement)
    conn.prepareStatement("INSERT INTO employees (empoyee_id, last_name,
address)
    VALUES (?, ?, ?)");
pstmt.setFormOfUse(3, oracle.jdbc.OraclePreparedStatement.FORM_NCHAR);
pstmt.setInt(1, employee_id);
pstmt.setCHAR(2, new oracle.sql.CHAR(ename, dbCharset));
pstmt.setCHAR(3, new oracle.sql.CHAR(eaddress, ncCharset));
pstmt.execute();
pstmt.close();
```

The oracle.sql.CHAR in Oracle Object Types

In Oracle9*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 9–1 does not apply to Oracle object access. Instead, the `oracle.sql.CHAR` class is used for passing SQL `CHAR` and SQL `NCHAR` data of an object type from the database to the client. The following is an example of an object type created using SQL:

```
CREATE TYPE person_type AS OBJECT (name VARCHAR2(30), address NVARCHAR(256), 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.CHAR address;
   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 `VARCHAR2` datatype. JDBC populates this class with the byte representation of the `VARCHAR2` data in the database and the

`CharacterSet` object corresponding to the database character set. The following code retrieves a `person` object from the `employees` 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;
oracle.sql.CHAR sql_address=p.address;
String java_name = sql_name.getString();
String java_name = sql_address.getString();
```

The `getString()` method of the `oracle.sql.CHAR` class converts the byte array from the database character set to UTF-16 by calling Oracle's Java data conversion classes and returning 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.

Restrictions on Accessing SQL CHAR Data with JDBC

This section contains the following topics:

- SQL CHAR Data Size Restriction With the JDBC Thin Driver
- Character Integrity Issues in a Multibyte Database Environment

SQL CHAR Data Size Restriction With the JDBC Thin Driver

If the database character set is neither ASCII (US7ASCII) nor ISO Latin1 (WE8ISO8859P1), then the JDBC thin driver must impose size restrictions for SQL `CHAR` bind parameters that are more restrictive than normal database size limitations. This is necessary to allow for data expansion during conversion.

The JDBC thin driver checks SQL `CHAR` bind sizes when a `set`*`XXX`*`()` method (except for the `setCharacterStream()` method) is called. If the data size exceeds the size restriction, then the driver returns a SQL exception (`SQLException: Data size bigger than max size for this type"`) from the `set`*`XXX`*`()` call. This limitation is necessary to avoid the chance of data corruption when conversion of character data occurs and increases the length of the data. This limitation is enforced in the following situations:

- Using the JDBC thin driver
- Using binds (not defines)
- Using SQL `CHAR` datatypes
- Connecting to a database whose character set is neither ASCII (US7ASCII) nor ISO Latin1 (WE8ISO8859P1)

When the database character set is neither US7ASCII nor WE8ISO8859P1, the JDBC thin driver converts Java UTF-16 characters to UTF-8 encoding bytes for SQL `CHAR` binds. The UTF-8 encoding bytes are then transferred to the database, and the database converts the UTF-8 encoding bytes to the database character set encoding.

This conversion to the character set encoding can result in an increase in the number of bytes required to store the data. The expansion factor for a database character set indicates the maximum possible expansion in converting from UTF-8 to the character set. If the database character set is either UTF8 or AL32UTF8, then the expansion factor (`exp_factor`) is `1`. Otherwise, the expansion factor is equal to the maximum character size (measured in bytes) in the database character set.

Table 9–2 shows the database size limitations for SQL `CHAR` data and the JDBC thin driver size restriction formulas for SQL `CHAR` binds. Database limits are in bytes. Formulas determine the maximum allowed size of the UTF-8 encoding in bytes.

Table 9–2 Maximum SQL CHAR Bind Sizes

Oracle Version	Datatype	Maximum Bind Size Allowed by Database	Formula for Determining the Maximum Bind Size, Measured in UTF-8 Bytes
Oracle8 and later	`CHAR`	2000 bytes	`4000/`*`exp_factor`*
Oracle8 and later	`VARCHAR2`	4000 bytes	`4000/`*`exp_factor`*
Oracle8 and later	`LONG`	2^{31} - 1 bytes	$(2^{31} - 1)/exp\_factor$

The formulas guarantee that after the data is converted from UTF-8 to the database character set, the size of the data will not exceed the maximum size allowed in the database.

The number of UTF-16 characters that can be supported is determined by the number of bytes per character in the data. All ASCII characters are one byte long in UTF-8 encoding. Other character types can be two or three bytes long.

Table 9–3 lists the expansion factors of some common server character sets. It also shows the JDBC thin driver maximum bind sizes for `CHAR` and `VARCHAR2` data for each character set.

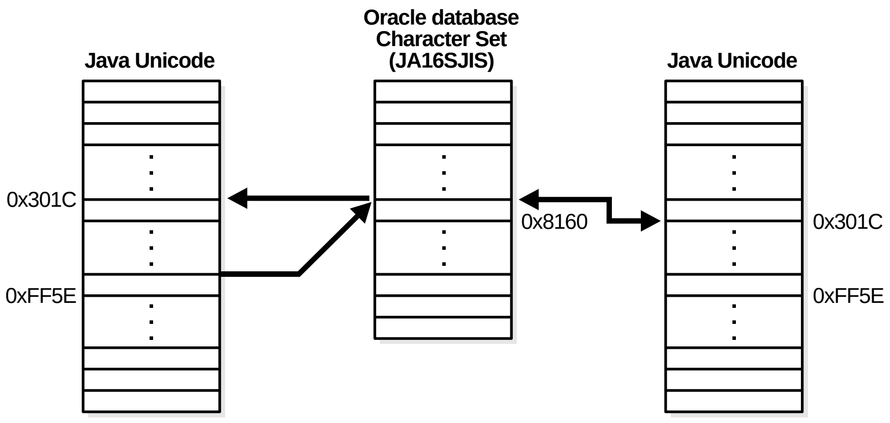
Table 9–3 Expansion Factor and Maximum Bind Size for Common Server Character Sets

Server Character Set	Expansion Factor	JDBC Thin Driver Maximum Bind Size for SQL CHAR Data, Measured in UTF-8 Bytes
WE8DEC	1	4000 bytes
JA16SJIS	2	2000 bytes
JA16EUC	3	1333 bytes
AL32UTF8	1	4000 bytes

Character Integrity Issues in a Multibyte Database Environment

Oracle JDBC drivers perform character set conversions as appropriate when character data is inserted into or retrieved from the database. The drivers convert Unicode characters used by Java clients to Oracle database character set characters, and vice versa. Character data that makes 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 is the Unicode full-width tilde character (0xFF5E) and its mapping to Oracle's JA16SJIS character set. The round trip conversion for this Unicode character results in the Unicode character 0x301C, which is a wave dash (a character commonly used in Japan to indicate range), not a tilde.

Figure 9–2 Character Integrity



This issue is not a bug in Oracle's JDBC. It is an unfortunate side effect of the ambiguity in character mapping specification on different operating systems. Fortunately, this problem affects only a small number of characters in a small number of Oracle character sets such as JA16SJIS, JA16EUC, ZHT16BIG5, and KO16KS5601. The workaround is to avoid making a full round-trip with these characters.

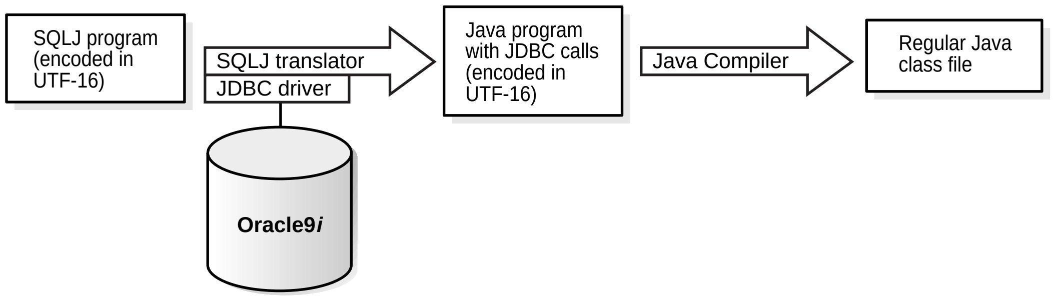
Globalization Support for SQLJ

SQLJ is a SQL-to-Java translator that translates embedded SQL statements in a Java program into the corresponding JDBC calls regardless of which JDBC driver is used. It also provides a callable interface that the Oracle9*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 9–3 shows how the SQLJ translator works. The figure is described in the following sections:

- Using Unicode Characters in SQLJ programs
- Using the oracle.sql.NString class

Figure 9–3 Using the SQLJ Translator


SQLJ program (encoded in UTF-16)
SQLJ translator
JDBC driver
Oracle9*i*
Java program with JDBC calls (encoded in UTF-16)
Java Compiler
Regular Java class file

Using Unicode Characters in SQLJ programs

SQLJ enables multilingual Java application development by allowing SQLJ files encoded in different encoding schemes (those supported by the JDK). In Figure 9–3, a UTF-16 encoded SQLJ program is being passed to the SQLJ translator and the Java program output is also encoded in UTF-16. SQLJ preserves the encoding of the source in the target. To specify the encoding of the source, use the `-encoding` option as follows:

```
sqlj -encoding Unicode source_file
```

Unicode notation \uXXXX (which is referred to as a Unicode escape sequence) can be used in embedded SQL statements for characters that cannot be represented in the encoding of the SQLJ program file. This enables you to specify multilingual object names in the SQL statement without using a UTF-16-encoded SQLJ file. The following SQLJ code shows the use of Unicode escape sequences in embedded SQL as well as in a string literal.

```
int employee_id = 12345;
String name last_name = "\uFF2A\uFF4F\uFF45";
double raise = 0.1;

#sql { INSERT INTO E\u006D\u0070 (last_name, employee_id) VALUES (:last_name,
:employee_id)};
#sql { UPDATE employees SET salary = :(getNewSal(raise, last_name))
WHERE last_name = :last_name};
```

See Also: "A Multilingual Demo Application in SQLJ" on page 9-23 for an example of SQLJ usage for a multilingual Java application

Using the oracle.sql.NString class

In Oracle9*i*, the `oracle.sql.NString` class is introduced in SQLJ to support the NVARCHAR2, NCHAR, and NCLOB Unicode datatypes. You can declare a bind on NCHAR column using a Java object of the `oracle.sql.NString` type, and use it in the embedded SQL statements in SQLJ programs.

```
int employee_id = 12345;
oracle.sql.NString last_name = new oracle.sql.NString ("\uFF2A\uFF4F\uFF45");
double raise = 0.1;
#sql { INSERT INTO E\u006D\u0070 (last_name, employee_id VALUES (:last_name,
:employee_id)};
#sql { UPDATE employees SET salary = :(getNewSal(raise, last_name)) = :last_
name};
```

This example binds the `last_name` object of the `oracle.sql.NString` datatype to the `last_name` database NVARCHAR2 column.

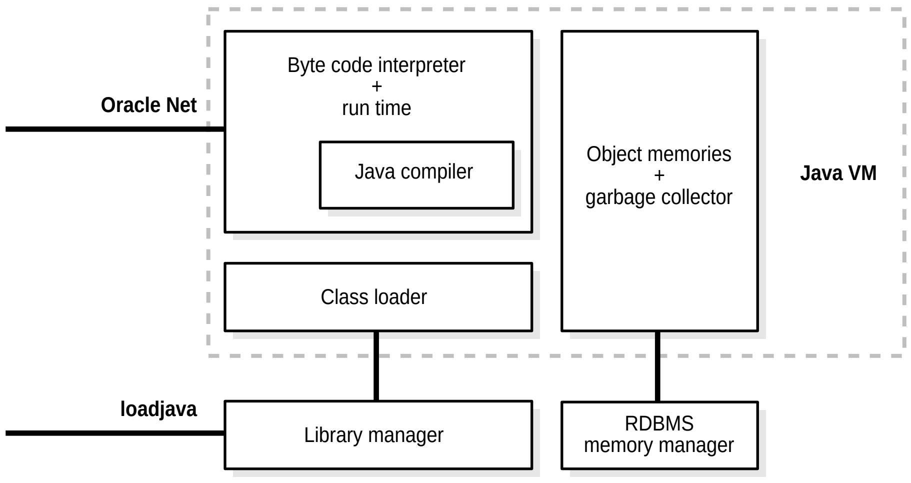
> **See Also:** "Binding and Defining Java Strings in Unicode" on page 6-26 for more details on the SQL NCHAR datatypes support in SQLJ

Globalization Support for Java Virtual Machine

The Oracle9*i* Java Virtual Machine (Java VM) is integrated into the database server to enable the running of Java classes stored in the database. Oracle9*i* enables you to store Java class files, Java or SQLJ source files, and Java resource files into the database. Then the Java entry points to SQL can be published so that Java can be called from SQL or PL/SQL and the Java byte code can be run.

In addition to the engine that interprets Java byte code, the Oracle Java VM includes the core runtime classes of the Java Development Kit (JDK). The components of the Java VM are depicted in Figure 9–4.

Figure 9–4 Components of Oracle's Java Virtual Machine



The Java VM provides:

- An embedded Java class loader that locates, loads, and initializes locally stored Java classes in the database
- A Java compiler that translates standard Java programs into standard Java `.class` binary representations

A library manager is also included to manage Java program, class, and resource files as schema objects known as **library units**. It not only loads and manages these Java files in the database, but also maps Java name space to library units. For example:

```
public class Greeting
{
   public String Hello(String name)
   {
     return ("Hello" + name + "!");
   }
}
```

After the preceding Java code is compiled, it is loaded into the database as follows:

```
loadjava Greeting.class
```

As a result, a library unit called `Greeting` is created as a schema object in the database.

Class and method names containing characters that cannot be represented in the database character set are handled by generating a US7ASCII library unit name and mapping it to the real class name stored in a `RAW` column. This enables the class loader to find the library unit corresponding to the real class name when Java programs run in the server. In other words, the library manager and the class loader support class names or method names outside the namespace of the database character set.

Globalization Support for Java Stored Procedures

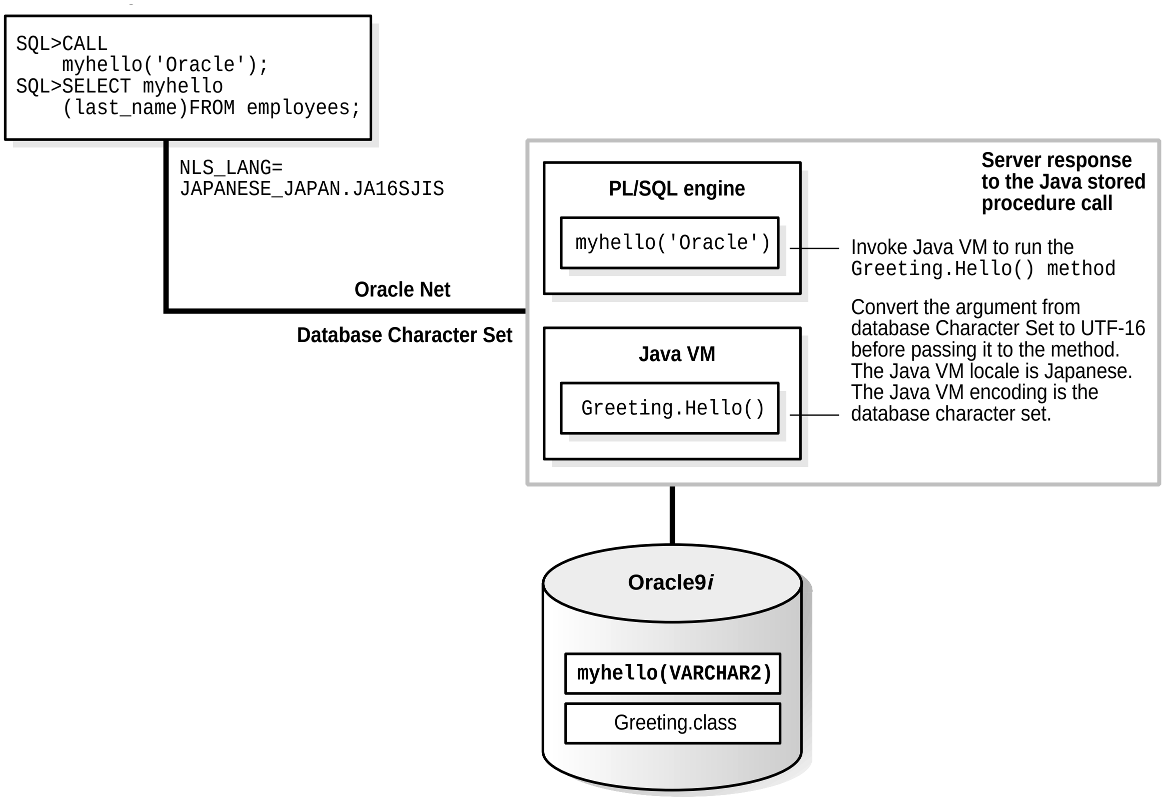
A Java stored procedure or function requires that the library unit of the Java classes implementing it already be present in the database. Using the `Greeting` library unit example in the previous section, the following call data definition language (DDL) publishes the method `Greeting.Hello()` as a 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 function. Users can call the Java stored procedures the same way they call any PL/SQL stored procedures.

Figure 9–5 depicts the runtime environment of a stored function.

Figure 9–5 Running Java Stored Procedures


SQL>CALL
myhello('Oracle');
SQL>SELECT myhello
(last_name)FROM employees;
NLS_LANG=
JAPANESE_JAPAN.JA16SJIS
Oracle Net
Database Character Set
PL/SQL engine
myhello('Oracle')
Java VM
Greeting.Hello()
Server response to the Java stored procedure call
Invoke Java VM to run the Greeting.Hello() method
Convert the argument from database Character Set to UTF-16 before passing it to the method. The Java VM locale is Japanese. The Java VM encoding is the database character set.
Oracle9i
myhello(VARCHAR2)
Greeting.class

The locale of the Java VM is Japanese and its encoding is the database character set. The client's `NLS_LANG` environment variable is defined as `JAPANESE_JAPAN.JA16SJIS`. Oracle Net converts the JA16SJIS characters in the client to the database character set characters if the characters are different.

The Java entry point, `Greeting.Hello()`, is called by invoking the proxy PL/SQL `myhello()` from the client. The server process serving the client runs as a normal PL/SQL stored function and uses the same syntax. The PL/SQL engine takes a call specification for a Java method and calls the Java VM. Next, it passes the method name of the Java stored function and the argument to the Java VM for execution. The Java VM takes control, calls the SQL to Java using code to convert the `VARCHAR2` argument from the database character set to UTF-16, loads the `Greeting` class, and runs the `Hello()` method 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:

- The strings in the arguments of Java stored procedures are automatically converted from SQL datatypes in the database character set to UTF-16-encoded Java strings.
- The default Java locale of the Java VM follows the language setting of the current database session derived from the `NLS_LANG` environment variable of the client. A mapping of Oracle language and territory names to Java locale names is in place for this purpose. In additions, the default encoding of the Java VM follows the database character set.
- 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 must be in the database character set because it must be published to SQL as DDL.

Configurations for Multilingual Applications

To develop and deploy multilingual Java applications for Oracle9*i*, the database configurations and client environments for the targeted systems must be determined.

This section contains the following topics:

- Configuring a Multilingual Database
- Globalization Support for Java Stored Procedures
- Clients with Different Languages

Configuring a Multilingual Database

In order to store multilingual data in an Oracle9*i* database, you need to configure the database appropriately. There are two ways to store Unicode data into the database:

- As SQL `CHAR` datatypes in a Unicode database
- As SQL `NCHAR` datatypes in a non-Unicode database

See Also: Chapter 5, "Supporting Multilingual Databases with Unicode" for more information about choosing a Unicode solution and configuring the database for Unicode

Globalization Support for Java Stored Procedures

For each Oracle9*i* session, a separate Java VM instance is created in the server for running the Java stored procedure, and Oracle9*i* Java support ensures that the locale of the Java VM instance is the same as that of the client Java VM. Hence the Java stored procedures always run on the same locale in the database as the client locale.

For non-Java clients, the default locale of the Java VM instance will be the Java locale that best corresponds to the `NLS_LANGUAGE` and `NLS_TERRITORY` session parameters propagated from the client `NLS_LANG` environment variable.

Internationalizing Java code

Java stored procedures are server objects which are accessible from clients of different language preferences. They should be internationalized so that they are sensitive to the Java locale of the Java VM, which is initialized to the locale of the client.

With JDK internationalization support, you can specify a Java locale object to any locale-sensitive methods or use the default Java locale of the Java VM for those methods. The following are examples of how to internationalize a Java stored procedure:

- Externalize all localizable strings or objects from the Java code to resource bundles and make the resource bundles as part of the procedure. Any messages returned from the resource bundle will be in the language of the client locale or whatever locale you specify.
- Use the Java formatting classes such as `DateFormat` and `NumberFormat` to format the date, time, numbers, and currencies with the assumption that they will reflect the locale of the calling client.
- Use Java locale-sensitive string classes such as `Character`, `Collator`, and `BreakIterator` to check the classification of a character, compare two strings linguistically, and parse a string character by character.

Transferring Multilingual Data

All Java server objects access the database with the JDBC server-side internal driver and should use either a Java string or `oracle.sql.CHAR` to represent string data to and from the database. Java strings are always encoded in UTF-16, and the required conversion from the database character set to UTF-16 is done transparently. `oracle.sql.CHAR` stores the database data in byte array and tags it with a character set ID. `oracle.sql.CHAR` should be used when no string manipulation is required on the data. For example, it is the best choice for transferring string data from one table to another in the database.

Clients with Different Languages

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

- Java applets running in browsers

  Java applets running in browsers can access the Oracle9*i* database through the JDBC thin driver. No client-side Oracle library is required. The applets use the JDBC thin driver to invoke SQL, PL/SQL as well as Java stored procedures. The JDBC thin driver makes sure that Java stored procedures run in the same locale as the Java VM running the applets.

- Dynamic HTML on browsers

  HTML pages invoke Java servlets by using URLs over HTTP. The Java servlets running in the middle tier construct dynamic HTML pages and deliver them back to the browser. They should determine the locale of a user and construct the page according to the language and cultural convention preferences of the user and use JDBC to connect to the database.

- Java applications running on client Java VMs

  Java applications running on the Java VM of the client machine can access the database through either JDBC OCI or JDBC thin drivers. Java applications can also be a middle tier servlet running on a Web server. The applications use JDBC drivers to invoke SQL, PL/SQL as well as Java stored procedures. The JDBC Thin and JDBC OCI drivers make sure that Java stored procedures will be running in the same locale as that of the client Java VM.

- C clients such as OCI, Pro*C/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.

A Multilingual Demo Application in SQLJ

This section contains a simple bookstore application written in SQLJ to demonstrate a database storing book information in different languages, and how SQLJ and JDBC are used to access the book information from the database. It also demonstrates the use of internationalized Java stored procedures to accomplish transactional tasks in the database server. The sample program consists of the following components:

- The SQLJ client Java application that displays a list of books in the store and allow users to add new books to and remove books from the inventory
- A Java stored procedure to add a new book to the inventory
- A Java stored procedure to remove an existing book from the inventory

This section contains the following topics:

- Database Schema for the Multilingual Demo Application
- Java Stored Procedures for the Multilingual Demo Application
- The SQLJ Client for the Multilingual Demo Application

Database Schema for the Multilingual Demo Application

AL32UTF8 is the database character set that is used to store book information, such as names and authors, in languages around the world.

The book table is described in Table 9–4.

Table 9–4 Columns in the book Table of the Multilingual Demo

Column Name	Datatype
ID (primary key)	NUMBER(10)
NAME	VARCHAR(300)
PUBLISH_DATE	DATE
AUTHOR	VARCHAR(120)

Table 9–4 Columns in the book Table of the Multilingual Demo (Cont.)

Column Name	Datatype
`PRICES`	`NUMBER(10,2)`

The `inventory` table is described in Table 9–5.

Table 9–5 Columns in the invertory Table of the Multilingual Demo

Column Name	Datatype
`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 improve performance during book searches. A `BOOKSEQ` sequence is be created to generate a unique book ID.

Java Stored Procedures for the Multilingual Demo Application

The Java class called `Book` is created to implement the methods `Book.remove()` and `Book.add()` that perform the tasks of removing books from and adding books to the inventory respectively. They are defined according to the following code. In this class, only the `remove()` method and the constructor are shown. The resource bundle `BookRes.class` is used to store localizable messages. The `remove()` method returns a message gotten from the resource bundle according to the current Java VM locale. There is no JDBC connection required to access the database because the stored procedure is already running in the context of a database session.

```
import java.sql.*;
import java.util.*;
import sqlj.runtime.ref.DefaultContext;
/* The book class implementation the transaction logics of the
   Java stored procedures.*/
public class Book
{
   static ResourceBundle rb;
   static int q, id;
   static DefaultContext ctx;
   public Book()
   {
     try
     {
         DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
         DefaultContext.setDefaultContext(ctx);
         rb = java.util.ResourceBundle.getBundle("BookRes");
      }
      catch (Exception e)
      {
        System.out.println("Transaction failed: " + e.getMessage());
      }
   }
  public static String remove(int id, int quantity, String location) throws
    SQLException
  {
      rb = ResourceBundle.getBundle("BookRes");
      try
      {
        #sql {SELECT QUANTITY INTO :q FROM INVENTORY WHERE ID = :id AND
        LOCATION = :location};
        if (id == 1) return rb.getString ("NotEnough");
       }
      catch (Exception e)
      {
          return rb.getString ("NotEnough");
      }
      if ((q - quantity) == 0)
      {
          #sql {DELETE FROM INVENTORY WHERE ID = :id AND LOCATION = :location};
          try
          {
              #sql {SELECT SUM(QUANTITY) INTO :q FROM INVENTORY WHERE ID = :id};
          }
          catch (Exception e)
          {
              #sql { DELETE FROM BOOK WHERE ID = :id };
              return rb.getString("RemoveBook");
          }
          return rb.getString("RemoveInventory");
      }
      else
      {
         if ((q-quantity) < 0) return rb.getString ("NotEnough");
         #sql { UPDATE INVENTORY SET QUANTITY = :(q-quantity) WHERE ID = :id and
         LOCATION = :location };
         return rb.getString("DecreaseInventory");
```

```
        }
    }
    public static String add( String bname, String author, String location,
           double price, int quantity, String publishdate )  throws SQLException
    {
       rb = ResourceBundle.getBundle("BookRes");
       try
       {
           #sql { SELECT ID into :id FROM BOOK WHERE NAME = :bname AND AUTHOR =
          :author };
       }
       catch (Exception e)
       {
           #sql { SELECT BOOKSEQ.NEXTVAL INTO :id FROM DUAL };
           #sql { INSERT INTO BOOK VALUES (:id, :bname,
           TO_DATE(:publishdate,'YYYY-MM-DD'), :author, :price) };
           #sql { INSERT INTO INVENTORY VALUES (:id, :location, :quantity) };
           return rb.getString("AddBook");
       }
       try
       {
           #sql { SELECT QUANTITY INTO :q FROM INVENTORY WHERE ID = :id
           AND LOCATION = :location };
       }
       catch (Exception e)
       {
            #sql { INSERT INTO INVENTORY VALUES (:id, :location, :quantity) };
            return rb.getString("AddInventory");
       }
       #sql { UPDATE INVENTORY SET QUANTITY = :(q + quantity) WHERE ID = :id
       AND LOCATION = :location };
       return rb.getString("IncreaseInventory");
    }
  }
```

After the `Book.remove()` and `Book.add()` methods are defined, they are in turn published as Java stored functions in the database called `REMOVEBOOK()` and `ADDBOOK()`:

```
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 is first 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 set is not AL32UTF8 or UTF8, then any Unicode characters in the Java strings that cannot be represented in the database character set will be replaced by a replacement character. 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 for the Multilingual Demo Application

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. It is defined in the following code.

A `BookDB` object is created when the sample program starts up with the user name, password, and the location of the database. The methods are called from the GUI portion of the applications. The `removeBook()` and `addBook()` methods 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 `books` iterator inside the `BookDB` object. (The `BookRecs` class is generated by SQLJ.) 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;
            }
        }
```

10

Character Set Migration

This chapter discusses character set conversion and character set migration. It includes the following topics:

Overview of Character Set Migration

Choosing the appropriate character set for your database is an important decision. When you choose the database character set, consider the following factors:

- The type of data you need to store
- The languages that the database needs to accommodate now and in the future
- The different size requirements of each character set and the corresponding performance implications

A related topic is choosing a new character set for an existing database. Changing the database character set for an existing database is called **character set migration**. Migrating from one database character set to another involves additional considerations beyond choosing a character set for a new database. Plan character set migration to minimize data loss from:

- Data Truncation
- Character Set Conversion Issues

See Also: Chapter 2, "Choosing a Character Set"

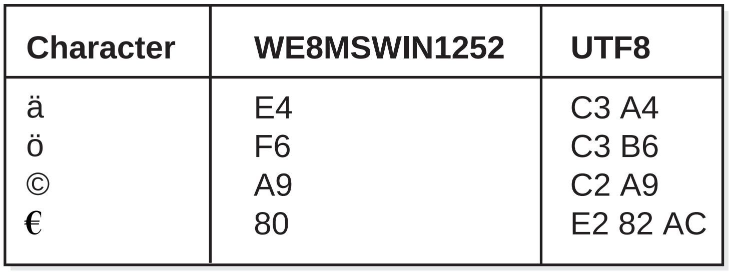
Data Truncation

When the database is created using byte semantics, the sizes of the `CHAR` and `VARCHAR2` datatypes are specified in bytes, not characters. For example, the specification `CHAR(20)` in a table definition allows 20 bytes for storing character data. This is acceptable when the database character set uses a single-byte character encoding scheme because the number of characters is equivalent to the number of bytes. If the database character set uses a multibyte character encoding scheme, then the number of bytes no longer equals the number of characters because a character can consist of one or more bytes.

During migration to a new character set, it is important to verify the column widths of existing `CHAR` and `VARCHAR2` columns because they might need to be extended to support an encoding that requires multibyte storage. Truncation of data can occur if conversion causes expansion of data.

Figure 10–1 shows an example of data expansion when single-byte characters become multibyte. For example, ä (a with an umlaut) is a single-byte character in WE8MSWIN1252, but it becomes a two-byte character in UTF8. Also, the Euro symbol expands from one byte to three bytes.

Figure 10–1 Single-Byte and Multibyte Encoding



Character	WE8MSWIN1252	UTF8
ä	E4	C3 A4
ö	F6	C3 B6
©	A9	C2 A9
€	80	E2 82 AC

The maximum number of bytes for CHAR and VARCHAR2 datatypes is 2000 and 4000, respectively. If the data in the new character set requires columns that are wider than 2000 and 4000 bytes for CHAR and VARCHAR2 datatypes, then you need to change your schema.

See Also: "Length Semantics" on page 2-12

Additional Problems Caused by Data Truncation

Data truncation can cause the following problems:

- In the database data dictionary, schema object names cannot exceed 30 bytes in length. Schema objects are tables, clusters, views, indexes, synonyms, tablespaces, and usernames. You must rename schema objects if their names exceed 30 bytes in the new database character set. For example, one Thai character in the Thai national character set requires 1 byte. In UTF8, it requires 3 bytes. If you have defined a table whose name is 11 Thai characters, then the table name must be shortened to 10 or fewer Thai characters when you change the database character set to UTF8.
- If existing Oracle usernames or passwords are created based on characters that will change in size in the new character set, users will experience login difficulties due to authentication failures after the migration to a new character set. This is because the encrypted usernames and passwords stored in the data dictionary are not updated during migration to a new character set. For example, if the current database character set is WE8MSWIN1252 and the new database character set is UTF8, then the length of the username scött (o with an umlaut) will change from 5 bytes to 6 bytes. In UTF8, scött will no longer be able to log in because of the difference in the username. Oracle Corporation recommends that usernames and passwords be based on ASCII characters. If

they are not, you must reset the affected usernames and passwords after migrating to a new character set.

- When CHAR data contains characters that will be expanded after migration to a new character set, space padding will not be removed during database export by default. This means that these rows will be rejected upon import into the database with the new character set. The workaround is to set the BLANK_TRIMMING initialization parameter to TRUE before importing the CHAR data.

  See Also: *Oracle9i Database Reference* for more information about the BLANK_TRIMMING initialization parameter

Character Set Conversion Issues

This section includes the following topics:

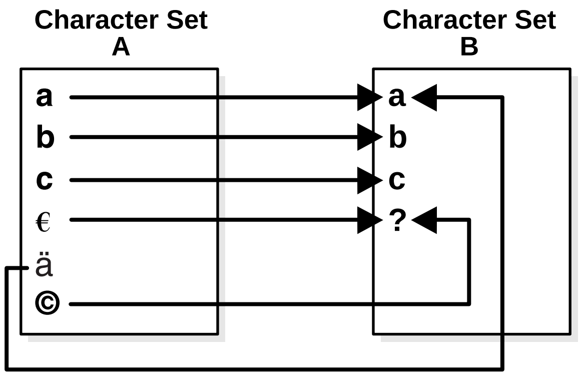
- Replacement Characters that Result from Using the Export and Import Utilities
- Invalid Data That Results from Setting the Client's NLS_LANG Parameter Incorrectly

Replacement Characters that Result from Using the Export and Import Utilities

The Export and Import utilities can convert character sets from the original database character set to the new database character set. However, character set conversions can sometimes cause data loss or data corruption. For example, if you are migrating from character set A to character set B, the destination character set B should be a superset of character set A. The destination character, B, is a **superset** if it contains all the characters defined in character set A. Characters that are not available in character set B are converted to replacement characters, which are often specified as ? or ¿ or a character that is related to the unavailable character. For example, ä (a with an umlaut) can be replaced by a. Replacement characters are defined by the target character set.

Figure 10–2 shows an example of a character set conversion in which the copyright and Euro symbols are converted to ? and ä is converted to a.

Figure 10–2 Replacement Characters in Character Set Conversion



To reduce the risk of losing data, choose a destination character set with a similar character repertoire. Migrating to Unicode can be an attractive option because UTF8 contains characters from most legacy character sets.

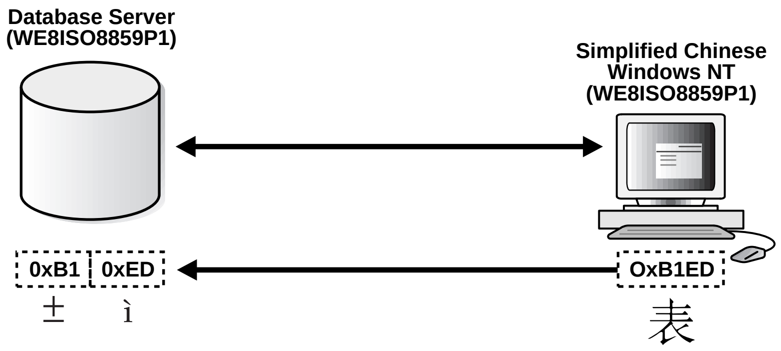
Invalid Data That Results from Setting the Client's NLS_LANG Parameter Incorrectly

Another character set migration scenario that can cause the loss of data is migrating a database that contains invalid data. Invalid data usually occurs in a database because the `NLS_LANG` parameter is not set properly on the client. The `NLS_LANG` value should reflect the client operating system code page. For example, in an English Windows environment, the code page is WE8MSWIN1252. When the `NLS_LANG` parameter is set properly, the database can automatically convert incoming data from the client operating system. When the `NLS_LANG` parameter is not set properly, then the data coming into the database is not converted properly. For example, suppose that the database character set is UTF8, the client is an English Windows operating system, and the `NLS_LANG` setting on the client is UTF8. Data coming into the database is encoded in WE8MSWIN1252 and is not converted to UTF8 data because the `NLS_LANG` setting on the client matches the database character set. Thus Oracle assumes that no conversion is necessary, and invalid data is entered into the database.

This can lead to two possible data inconsistency problems. One problem occurs when a database contains data from a character set that is different from the database character set but the same code points exist in both character sets. For example, if the database character set is WE8ISO8859P1 and the `NLS_LANG` setting of the Chinese Windows NT client is `SIMPLIFIED CHINESE_`

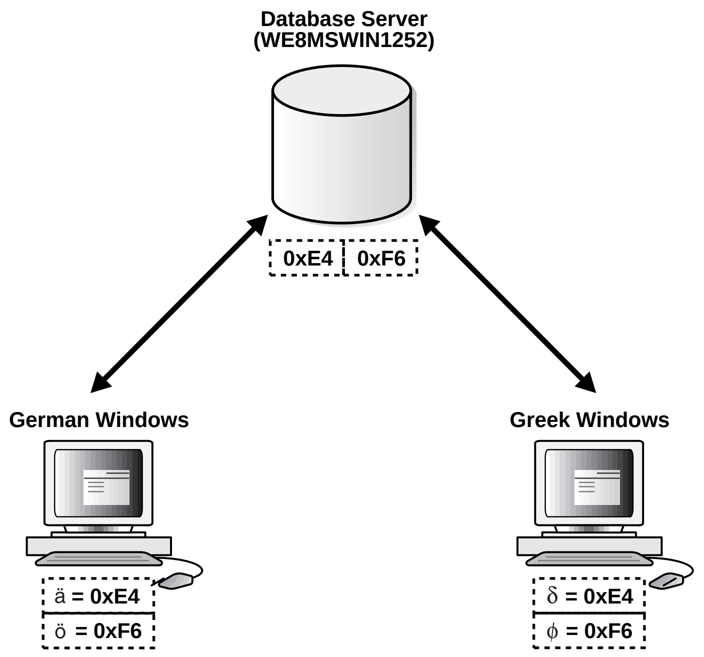
CHINA.WE8ISO8859P1, then all multibyte Chinese data (from the ZHS16GBK character set) is stored as multiples of single-byte WE8ISO8859P1 data. This means that Oracle will treat these characters as single-byte WE8ISO8859P1 characters. Hence all SQL string manipulation functions such as SUBSTR or LENGTH will be based on bytes rather than characters. All bytes constituting ZHS16GBK data are legal WE8ISO8859P1 codes. If such a database is migrated to another character set, for example, UTF8, character codes will be converted as if they were in WE8ISO8859P1. This way, each of the two bytes of a ZHS16GBK character will be converted separately, yielding meaningless values in UTF8. Figure 10–3 shows an example of this incorrect character set replacement.

Figure 10–3 Incorrect Character Set Replacement



The second possible problem is having data from mixed character sets inside the database. For example, if the data character set is WE8MSWIN1252, and two separate Windows clients using German and Greek are both using the NLS_LANG character set setting as WE8MSWIN1252, then the database will contain a mixture of German and Greek characters. Figure 10–4 shows how different clients can use different character sets in the same database.

Figure 10–4 Mixed Character Sets



For database character set migration to be successful, both of these cases require manual intervention because Oracle cannot determine the character sets of the data being stored.

Changing the Database Character Set of an Existing Database

Database character set migration has two stages: data scanning and data conversion. Before you change the database character set, you need to identify possible database character set conversion problems and truncation of data. This step is called **data scanning**.

Data scanning identifies the amount of effort required to migrate data into the new character encoding scheme before changing the database character set. Some examples of what may be found during a data scan are the number of schema objects where the column widths need to be expanded and the extent of the data that does not exist in the target character repertoire. This information helps to determine the best approach for converting the database character set.

There are three approaches to converting data from one database character set to another if the database does not contain any of the inconsistencies described in "Character Set Conversion Issues" on page 10-4. A description of methods to migrate databases with such inconsistencies is out of the scope of this documentation. For more information, contact Oracle Consulting Services for assistance.

The approaches are:

- Migrating Character Data Using a Full Export and Import
- Migrating Character Data Using the ALTER DATABASE CHARACTER SET Statement
- Migrating Character Data Using the ALTER DATABASE CHARACTER SET Statement and Selective Imports

> **See Also:** Chapter 11, "Character Set Scanner" for more information about data scanning

Migrating Character Data Using a Full Export and Import

In most cases, a full export and import is recommended to properly convert all data to a new character set. It is important to be aware of data truncation issues, because columns with character datatypes may need to be extended before the import to handle an increase in size. Existing PL/SQL code should be reviewed to ensure that all byte-based SQL functions such as `LENGTHB`, `SUBSTRB`, and `INSTRB`, and PL/SQL `CHAR` and `VARCHAR2` declarations are still valid.

> **See Also:** *Oracle9i Database Utilities* for more information about the Export and Import utilities

Migrating Character Data Using the ALTER DATABASE CHARACTER SET Statement

The `ALTER DATABASE CHARACTER SET` statement is the fastest way to migrate a character set, but it can be used only under special circumstances. The `ALTER DATABASE CHARACTER SET` statement does not perform any data conversion, so it can be used if and only if the new character set is a strict superset of the current character set.

The new character set is a strict superset of the current character set if:

- Each and every character in the current character set is available in the new character set.

- Each and every character in the current character set has the same code point value in the new character set. For example, US7ASCII is a strict subset of many character sets.

Another restriction of the ALTER DATABASE CHARACTER SET statement is that it can be used only when the character set migration is between two single-byte character sets or between two multibyte character sets. If the planned character set migration is from a single-byte character set to a multibyte character set, then use the Export and Import utilities.

This restriction on using the `ALTER DATABASE CHARACTER SET` statement arises because of `CLOB` data. In Oracle9*i*, some internal fields in the data dictionary are stored in `CLOB` columns. Customers may also store data in `CLOB` fields. When the database character set is multibyte, `CLOB` data in Oracle9*i* is stored as UCS-2 data (two-byte, fixed-width Unicode). When the database character set is single-byte, `CLOB` data is stored using the database character set. Because the `ALTER DATABASE CHARACTER SET` statement does not convert data, `CLOB` columns remain in the original database character set encoding when the database character set is migrated from single-byte to multibyte. This introduces data inconsistency in the `CLOB` columns.

The syntax of the `ALTER DATABASE CHARACTER SET` statement is as follows:

```
ALTER DATABASE [db_name] CHARACTER SET new_character_set;
```

`db_name` is optional. The character set name should be specified without quotes. For example:

```
ALTER DATABASE CHARACTER SET AL32UTF8;
```

To change the database character set, perform the following steps:

1. Shut down the database, using either a `SHUTDOWN IMMEDIATE` or a `SHUTDOWN NORMAL` statement.
2. Do a full backup of the database because the `ALTER DATABASE CHARACTER SET` statement cannot be rolled back.
3. Complete the following statements:

```
STARTUP MOUNT;
ALTER SYSTEM ENABLE RESTRICTED SESSION;
ALTER SYSTEM SET JOB_QUEUE_PROCESSES=0;
ALTER SYSTEM SET AQ_TM_PROCESSES=0;
ALTER DATABASE OPEN;
ALTER DATABASE CHARACTER SET new_character_set;
```

```
SHUTDOWN IMMEDIATE; -- or SHUTDOWN NORMAL;
STARTUP;
```

See Also:

- *Oracle9i SQL Reference* for more information about the ALTER DATABASE CHARACTER SET statement
- Appendix A, "Locale Data" for a list of all superset character sets

Using the ALTER DATABASE CHARACTER SET Statement in an Oracle9*i* Real Application Clusters Environment

In a Oracle9*i* Real Application Clusters environment, ensure that no other Oracle background processes are running, with the exception of the background processes associated with the instance through which a user is connected, before attempting to issue the ALTER DATABASE CHARACTER SET statement. Use the following SQL statement to verify the environment:

```
SELECT SID, SERIAL#, PROGRAM FROM V$SESSION;
```

Set the CLUSTER_DATABASE initialization parameter to FALSE to allow the character set change to be completed. This is required in an Oracle9*i* Real Application Cluster environment; an exclusive startup is not sufficient.

Migrating Character Data Using the ALTER DATABASE CHARACTER SET Statement and Selective Imports

Another approach to migrating character data is to perform an ALTER DATABASE CHARACTER SET statement followed by selective imports. This method is best suited for a known distribution of convertible data that is stored within a small number of tables. A full export and import is too expensive in this scenario. For example, suppose you have a 100GB database with over 300 tables, but only 3 tables require character set conversions. The rest of the data is of the same encoding as the destination character set. The 3 tables can be exported and imported back to the new database after issuing the ALTER DATABASE CHARACTER SET statement.

Incorrect data conversion can lead to data corruption, so perform a full backup of the database before attempting to migrate the data to a new character set.

Migrating to the Oracle9*i* NCHAR Datatypes

In Oracle9*i*, data that is stored in columns of the NCHAR datatypes is stored exclusively in a Unicode encoding regardless of the database character set. This allows users to store Unicode in a database that does not use Unicode as the database character set.

This section includes the following topics:

- Migrating Oracle8 NCHAR Columns to Oracle9i
- Changing the National Character Set
- Migrating CHAR Columns to NCHAR Columns in an Oracle9i Database

Migrating Oracle8 NCHAR Columns to Oracle9*i*

In release 8.0, the Oracle Server introduced a national character datatype (NCHAR) that allows a second, alternate character set in addition to the database character set. The NCHAR datatypes support several fixed-width Asian character sets that were introduced to provide better performance when processing Asian character data.

In Oracle9*i*, the SQL NCHAR datatypes are limited to Unicode character set encoding (UTF8 and AL16UTF16). Any other Oracle8 Server character sets that were available for the NCHAR datatype, including Asian character sets such as JA16SJISFIXED are no longer supported.

The steps for migrating existing NCHAR, NVARCHAR2, and NCLOB columns to Oracle9*i* NCHAR datatypes are as follows:

1. Export all NCHAR columns from the Oracle8 or Oracle8*i* database.
2. Drop the NCHAR columns.
3. Upgrade database to Oracle9*i*.
4. Import the NCHAR columns into Oracle9*i*.

The Oracle9*i* migration utility can also convert Oracle8 and Oracle8*i* NCHAR columns to 9*i* NCHAR columns. A SQL NCHAR upgrade script called `utlchar.sql` is supplied with the migration utility. Run it at the end of the database migration to convert Oracle8 and Oracle8*i* NCHAR columns to the Oracle9*i* NCHAR columns. After the script has been executed, the data cannot be downgraded. The only way to move back to Oracle8 or Oracle8*i* is to drop all NCHAR columns, downgrade the database, and import the old NCHAR data from a previous Oracle8 or Oracle8*i* export file. Ensure that you have a backup (export file) of Oracle8 or Oracle8*i* NCHAR data, in case you need to downgrade your database in the future.

See Also:

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

Changing the National Character Set

To change the national character set, use the ALTER DATABASE NATIONAL CHARACTER SET statement. The syntax of the statement is as follows:

```
ALTER DATABASE [db_name] NATIONAL CHARACTER SET new_NCHAR_character_set;
```

db_name is optional. The character set name should be specified without quotes.

You can issue the ALTER DATABASE CHARACTER SET and ALTER DATABASE NATIONAL CHARACTER SET statements together if desired.

See Also: *Oracle9i SQL Reference* for the syntax of the ALTER DATABASE NATIONAL CHARACTER SET statement

Migrating CHAR Columns to NCHAR Columns in an Oracle9*i* Database

You can change a column's datatype definition using the following methods:

- The ALTER TABLE MODIFY statement
- Online table redefinition

The ALTER TABLE MODIFY statement has the following advantages over online table redefinition:

- Easier to use
- Fewer restrictions

Online table redefinition has the following advantages over the ALTER TABLE MODIFY statement:

- Faster for columns with a large amount of data
- Can migrate several columns at one time
- Table is available for DML during most of the migration process

- Avoids table fragmentation, which saves space and allows faster access to data.
- Can be used for migration from the CLOB datatype to the NCLOB datatype

This section contains the following topics:

- Using the ALTER TABLE MODIFY Statement to Change CHAR Columns to NCHAR Columns
- Using Online Table Redefinition to Migrate a Large Table to Unicode

Using the ALTER TABLE MODIFY Statement to Change CHAR Columns to NCHAR Columns

The ALTER TABLE MODIFY statement can be used to change table column definitions from the CHAR datatypes to NCHAR datatypes. It also converts all of the data in the column from the database character set to the NCHAR character set. The syntax of the ALTER TABLE MODIFY statement is as follows:

```
ALTER TABLE table_name MODIFY (column_name datatype);
```

If indexes have been built on the migrating column, then dropping the indexes can improve the performance of the ALTER TABLE MODIFY statement because indexes are updated when each row is updated.

The maximum column lengths for NCHAR and NVARCHAR2 columns are 2000 and 4000 bytes. When the NCHAR character set is AL16UTF16, the maximum column lengths for NCHAR and NVARCHAR2 columns are 1000 and 2000 characters, which are 2000 and 4000 bytes. If this size limit is violated during migration, consider changing the column to the NCLOB datatype instead.

Note: CLOB columns cannot be migrated to NCLOB columns using the ALTER TABLE MODIFY statement. Use online table redefinition to change a column from the CLOB datatype to the NCLOB datatype.

See Also: "Using Online Table Redefinition to Migrate a Large Table to Unicode" on page 10-13

Using Online Table Redefinition to Migrate a Large Table to Unicode

It takes significant time to migrate a large table with a large number of rows to Unicode datatypes. During the migration, the column data is unavailable for both reading and updating. Online table redefinition can significantly reduce migration time. Using online table redefinition also allows the table to be accessible to DML during most of the migration time.

Perform the following tasks to migrate a table to Unicode datatypes using online table redefinition:

1. Use the DBMS_REDEFINITION.CAN_REDEF_TABLE PL/SQL procedure to verify that the table can be redefined online. For example, to migrate the scott.emp table, enter the following command:

   ```
   DBMS_REDEFINITION.CAN_REDEF_TABLE('scott','emp');
   ```

2. Create an empty interim table in the same schema as the table that is to be redefined. Create it with NCHAR datatypes as the attributes. For example, enter a statement similar to the following:

   ```
   CREATE TABLE int_emp(
       empno NUMBER(4),
       ename NVARCHAR2(10),
       job NVARCHAR2(9),
       mgr NUMBER(4),
       hiredate DATE,
       sal NUMBER(7,2),
       deptno NUMBER(2),
       org NVARCHAR2(10));
   ```

3. Start the online table redefinition. Enter a command similar to the following:

   ```
   DBMS_REDEFINITION.START_REDEF_TABLE('scott',
   'emp',
   'int_emp',
   'empno empno,
   to_nchar(ename) ename,
   to_nchar(job) job,
   mgr mgr,
   hiredate hiredate,
   sal sal,
   deptno deptno,
   to_nchar(org) org');
   ```

   If you are migrating CLOB columns to NCLOB columns, then use the TO_NCLOB SQL conversion function instead of the TO_NCHAR SQL function.

4. Create triggers, indexes, grants, and constraints on the interim table. Referential constraints that apply to the interim table (the interim table is a parent or child table of the referential constraint) must be created in DISABLED mode. Triggers

that are defined on the interim table are not executed until the online table redefinition process has been completed.

5. You can synchronize the interim table with the original table. If many DML operations have been applied to the original table since the online redefinition began, then execute the DBMS_REDEFINITION.SYNC_INTERIM_TABLE procedure. This reduces the time required for the DBMS_REDEFINITION.FINISH_REDEF_TABLE procedure. Enter a command similar to the following:

   ```
   DBMS_REDEFINITION.SYNC_INTERIM_TABLE('scott', 'emp', 'int_emp');
   ```

6. Execute the DBMS_REDEFINITION.FINISH_REDEF_TABLE procedure. Enter a command similar to the following:

   ```
   DBMS_REDEFINITION.RINISH_REDEF_TABLE('scott', 'emp', 'int_emp');
   ```

   When this procedure has been completed, the following conditions are true:

   - The original table is redefined so that it has all the attributes, indexes, constraints, grants, and triggers of the interim table.
   - The referential constraints that apply to the interim table apply to the redefined original table.

7. Drop the interim table. Enter a statement similar to the following:

   ```
   DROP TABLE int_emp;
   ```

The results of the online table redefinition tasks are as follows:

- The original table is migrated to Unicode columns.
- The triggers, grants, indexes, and constraints defined on the interim table after the START_REDEF_TABLE subprogram and before the FINISH_REDEF_TABLE subprogram are defined for the redefined original table. Referential constraints that apply to the interim table now apply to the redefined original table and are enabled.
- The triggers, grants, indexes, and constraints defined on the original table before redefinition are transferred to the interim table and are dropped when you drop the interim table. Referential constraints that applied to the original table before redefinition were applied to the interim table and are now disabled.
- PL/SQL procedures and cursors that were defined on the original table before redefinition are invalidated. They are automatically revalidated the next time they are used. Revalidation may fail because the table definition has changed.

See Also: *Oracle9i Database Administrator's Guide* for more information about online table redefinition

Tasks to Recover Database Schema After Character Set Migration

You may need to perform additional tasks to recover a migrated database schema to its original state. Consider the issues described in Table 10–1.

Table 10–1 Issues During Recovery of a Migrated Database Schema

Issue	Description
Indexes	When table columns are changed from `CHAR` datatypes to `NCHAR` datatypes by the `ALTER TABLE MODIFY` statement, indexes that are built on the columns are changed automatically by the database. This slows down performance for the `ALTER TABLE MODIFY` statement. If you drop indexes before issuing the `ALTER TABLE MODIFY` statement, then re-create them after migration.
Constraints	If you disable constraints before migration, then re-enable them after migration.
Triggers	If you disable triggers before migration, then re-enable them after migration.
Replication	If the columns that are migrated to Unicode datatypes are replicated across several sites, then the changes should be executed at the master definition site. Then they will be propagated to the other sites.
Binary order	The migration from `CHAR` datatypes to `NCHAR` datatypes involves character set conversion if the database and `NCHAR` data have different character sets. The binary order of the same data in different encodings can be different. This affects applications that rely on binary order.

11 Character Set Scanner

This chapter introduces the Character Set Scanner, a globalization support utility for checking data before migrating character sets. The topics in this chapter include:

What is the Character Set Scanner?

The Character Set Scanner provides an assessment of the feasibility and potential issues in migrating an Oracle database to a new database character set. The Character Set Scanner checks all character data in the database and tests for the effects and problems of changing the character set encoding. At the end of the scan, it generates a summary report of the database scan. This report shows the scope work required to convert the database to a new character set.

Based on the information in the summary report, you can decide on the most appropriate method to migrate the database's character set. The methods are:

- Export and Import utilities
- ALTER DATABASE CHARACTER SET statement
- ALTER DATABASE CHARACTER SET statement with selective Export and Import

---

Note: If there are conversion exceptions reported by the Character Set Scanner, these problems must be fixed first before using any of the described methods to do the conversions. This may involve modifying the problem data to eliminate those exceptions. In extreme cases, both database and application might need to be modified. Oracle Corporation recommends you contact Oracle Consulting Services for services on database character set migration.

See Also: "Changing the Database Character Set of an Existing Database" on page 10-7

Conversion Tests on Character Data

The Character Set Scanner reads the character data and tests for the following conditions on each data cell:

- Do character code points of the data cells change when converted to the new character set?
- Can the data cells be successfully converted to the new character set?
- Will the post-conversion data fit into the current column size?

The Character Set Scanner reads and tests for data in `CHAR`, `VARCHAR2`, `LONG`, `CLOB`, `NCHAR`, `NVARCHAR2`, and `NCLOB` columns only. The Character Set Scanner does not perform post-conversion column size testing for LONG, CLOB, and NCLOB columns.

Access Privileges

To use the Character Set Scanner, you must have DBA privileges on the Oracle database.

Restrictions

All the character-based data in CHAR, VARCHAR2, LONG, and CLOB columns is stored in the database character set, which is specified with the CREATE DATABASE statement when the database was first created. However, in some configurations, it is possible to store data in a different character set from the database character set either intentionally or unintentionally. This happens most often when the NLS_LANG character set is the same as the database character set, because in such cases Oracle sends and receives data as is, without any conversion or validation. But it can also happen if one of the two character sets is a superset of the other, in which case many of the code points appear as if they were not converted. For example, if NLS_LANG is set to WE8ISO8859P1 and the database character set is WE8MSWIN1252, all code points except the range 128-159 are preserved through the client/server conversion.

Although a database that contains data not in its database character set cannot be converted to another character set by the three methods listed in "What is the Character Set Scanner?" on page 11-2, you can still use the Character Set Scanner to test the effect of the conversion that would take place if the data were in the database character set.

The encoding for different character sets can use the same code point for different characters.There is no automatic method to detect what the intended character is. Most European character sets share liberal use of the 8-bit range to encode native characters, so it is very possible for a cell to be reported as convertible but for the wrong reasons.

For example, this can occur when the Character Set Scanner is used with the FROMCHAR parameter set to WE8MSWIN1252. This single-byte character set encodes a character in every available code point so that no matter what data is being scanned, the scanner always identifies a data cell as being available in the source character set.

When you set FROMCHAR, you are assuming that all character data is in that character set but that the Character Set Scanner is not able to accurately determine the validity. Set the FROMCHAR parameter carefully.

The Character Set Scanner does not support the scanning of the VARRAY collection type.

Database Containing Data From Two or More Character Sets

If a database contains data from more than one character set, the Character Set Scanner cannot accurately test the effects of changing the database character set on the database because it cannot differentiate character sets easily. If the data can be divided into two separate tables, one for each character set, then the Character Set Scanner can perform two single table scans to verify the validity of the data.

For each scan, a different value of the FROMCHAR parameter can be used to tell the Character Set Scanner to treat all target columns in the table as if they were in the specified character set.

Database Containing Data Not From the Database Character Set

If a database contains data not in the database character set, but still in only one character set, the Character Set Scanner can perform a full database scan. Use the FROMCHAR parameter to tell the Character Set Scanner what character set the data is in.

Scan Modes in the Character Set Scanner

The Character Set Scanner provides three modes of database scan:

- Full Database Scan
- User Scan
- Table Scan

Full Database Scan

The Character Set Scanner reads and verifies the character data of all tables belonging to all users in the database including the data dictionary (SYS user), and it reports on the effects of the simulated migration to the new database character set. It scans all schema objects including stored packages, procedures and functions, and object names.

To understand the feasibility of migration to a new database character set, you need to perform a full database scan.

User Scan

The Character Set Scanner reads and verifies character data of all tables belonging to the specified user and reports on the effects on the tables of changing the character set.

The Character Set Scanner does not test for table definitions such as table names and column names. To see the effects on the schema definitions, you need to perform a full database scan.

Table Scan

The Character Set Scanner reads and verifies the character data of the specified table, and reports the effects on the table of changing the character set.

The Character Set Scanner does not test for table definitions such as table name and column name. To see the effects on the schema definitions, you need to perform a full database scan.

Using The Character Set Scanner

This section describes how to use the Character Set Scanner, including the steps you need to perform before scanning and the procedures on how to invoke the Character Set Scanner. The topics discussed are:

- Before Using the Character Set Scanner
- Character Set Scanner Compatibility
- Invoking the Character Set Scanner
- Getting Online Help for the Character Set Scanner
- The Parameter File

Before Using the Character Set Scanner

To use the Character Set Scanner, you must run the `csminst.sql` script on the database that you plan to scan. The `csminst.sql` script needs to be run only once. The script performs the following tasks to prepare the database for scanning:

- Creates a user named `CSMIG`

- Assigns the necessary privileges to CSMIG
- Assigns the default tablespace to CSMIG
- Connects as CSMIG
- Creates the Character Set Scanner system tables under CSMIG

The SYSTEM tablespace is assigned to CSMIG by default, so you need to ensure there is sufficient storage space available in the SYSTEM tablespace before scanning the database. The amount of space required depends on the type of scan and the nature of the data in the database.

> **See Also:** "Storage and Performance Considerations in the Character Set Scanner" on page 11-29

You can modify the default tablespace for CSMIG by editing the csminst.sql script. Modify the following statement in csminst.sql to assign your preferred tablespace to CSMIG as follows:

```
ALTER USER csmig DEFAULT TABLESPACE tablespace_name;
```

Then run csminst.sql using these commands and SQL statements:

```
% cd $ORACLE_HOME/rdbms/admin
% sqlplus "system/manager as sysdba"
SQL> START csminst.sql
```

Character Set Scanner Compatibility

The Character Set Scanner is certified with Oracle databases on any platforms running under the same release except that you cannot mix ASCII-based and EBCDIC-based platforms. For example, the Oracle9*i* release 2 (9.2) versions of the Character Set Scanner on any ASCII-based client platforms are certified to run with any Oracle9*i* release 2 (9.2) databases on any ASCII-based platforms, while EBCDIC-based clients are certified to run with any Oracle9*i* database on EBCDIC platforms.

Oracle Corporation recommends that you run the Character Set Scanner in the same Oracle home as the database when possible.

Invoking the Character Set Scanner

You can invoke the Character Set Scanner by one of these methods:

- Using the parameter file

```
csscan system/manager PARFILE=filename
```

PARFILE is a file containing the Character Set Scanner parameters you typically use.

- Using the command line

  ```
  csscan system/manager full=y tochar=utf8 array=10240 process=3
  ```

- Using an interactive session

  ```
  csscan system/manager
  ```

  In an interactive session, the Character Set Scanner prompts you for the following parameters:

  ```
  FULL/TABLE/USER
  TOCHAR
  ARRAY
  PROCESS
  ```

  If you want to specify parameters that are not listed, you need to invoke the Character Set Scanner using either the parameter file or the command line.

Getting Online Help for the Character Set Scanner

The Character Set Scanner provides online help. Enter `csscan help=y` on the command line to invoke the help screen.

You can let the Character Set Scanner prompt you for parameters by entering the CSSCAN command followed by your username and password. For example:

```
CSSCAN SYSTEM/MANAGER
```

Alternatively, you can control how the Character Set Scanner runs by entering the CSSCAN command followed by various parameters. To specify parameters, use keywords. For example:

```
CSSCAN SYSTEM/MANAGER FULL=y TOCHAR=utf8 ARRAY=102400 PROCESS=3
```

The following is a list of keywords for the Character Set Scanner:

```
Keyword    Default Prompt Description
---------- ------- ------ -------------------------------------------------
USERID             yes    username/password
```

```
FULL       N       yes    scan entire database
USER               yes    user name of the table to scan
TABLE              yes    list of tables to scan
EXCLUDE                   list of tables to exclude from scan
TOCHAR             yes    new database character set name
FROMCHAR                  current database character set name
TONCHAR                   new NCHAR character set name
FROMNCHAR                 current NCHAR character set name
ARRAY      10240   yes    size of array fetch buffer
PROCESS    1       yes    number of scan process
MAXBLOCKS                 split table if larger than MAXBLOCKS
CAPTURE     N             capture convertible data
SUPPRESS                  suppress error log by N per table
FEEDBACK                  feedback progress every N rows
BOUNDARIES                list of column size boundaries for summary report
LASTRPT    N              generate report of the previous database scan
LOG        scan           base name of log files
PARFILE                   parameter file name
PRESERVE   N              preserve existing scan results
HELP       N              show help screen
```

The Parameter File

The parameter file enables you to specify Character Set Scanner parameters in a file where they can be easily modified or reused. Create a parameter file using any flat file text editor. The command line option `PARFILE=`*`filename`* tells the Character Set Scanner to read the parameters from a specified file rather than from the command line. For example:

```
csscan parfile=filename
```

or

```
csscan username/password parfile=filename
```

The syntax for parameter file specifications is one of the following:

```
KEYWORD=value
KEYWORD=(value1, value2, ...)
```

The following is an example of a parameter file:

```
USERID=system/manager
USER=HR # scan HR's tables
TOCHAR=utf8
ARRAY=40960
```

```
PROCESS=2  # use two concurrent scan processes
FEEDBACK=1000
```

You can add comments to the parameter file by preceding them with the pound (#) sign. All characters to the right of the pound sign are ignored.

Character Set Scanner Parameters

The following topics are included in this section:

ARRAY Character Set Scanner Parameter

Default value:	`10240`
Minimum value:	`4096`
Maximum value:	Unlimited

Purpose:	Specifies the size in bytes of the array buffer used to fetch data. The size of the array buffer determines the number of rows fetched by the Character Set Scanner at any one time.

The following formula estimates the number of rows fetched at a time:

```
(rows in array) =
(ARRAY buffer size) / (sum of the CHAR and VARCHAR2 column sizes of a given table)
```

If the sum of the `CHAR` and `VARCHAR2` column sizes exceeds the array buffer size, then the Character Set Scanner fetches only one row at a time. Tables with `LONG`, `CLOB`, or `NCLOB` columns are fetched only one row at a time.

This parameter affects the duration of a database scan. In general, the larger the size of the array buffer, the shorter the duration time. Each scan process will allocate the specified size of array buffer.

BOUNDARIES Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies the list of column boundary sizes that are used for an application data conversion summary report. This parameter is used to locate the distribution of the application data for the `CHAR`, `VARCHAR2`, `NCHAR`, and `NVARCHAR2` datatypes.

For example, if you specify a `BOUNDARIES` value of (10, 100, 1000), then the application data conversion summary report produces a breakdown of the `CHAR` data into the following groups by their column length, `CHAR(1..10)`, `CHAR(11..100)` and `CHAR(101..1000)`. The behavior is the same for the `VARCHAR2`, `NCHAR`, and `NVARCHAR2` datatypes.

CAPTURE Character Set Scanner Parameter

Default value:	`N`
Range of values:	`Y` or `N`

Purpose:	Indicates whether to capture the information on the individual convertible rows as well as the default of storing the exception rows. The convertible rows information is written to the `CSM$ERRORS` table if the `CAPTURE` parameter is set to Y. This information can be used to deduce which records need to be converted to the target character set by selective export and import.

EXCLUDE Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies the names of the tables to be excluded from the scan

When this parameter is specified, the Character Set Scanner excludes the specified tables from the scan. You can specify the following when you specify the name of the table:

- *`schemaname`* specifies the name of the user's schema from which to exclude the table
- *`tablename`* specifies the name of the table or tables to be excluded

For example, the following command scans all of the tables that belong to the `hr` sample schema except for the `employees` and `departments` tables:

```
cssan system/manager USER=HR EXCLUDE=(HR.EMPLOYEES , HR.DEPARTMENTS) ...
```

FEEDBACK Character Set Scanner Parameter

Default value:	None
Minimum value:	`100`
Maximum value:	`100000`
Purpose:	Specifies that the Character Set Scanner should display a progress meter in the fort of a dot for every N number of rows scanned

For example, if you specify `FEEDBACK=1000`, then the Character Set Scanner displays a dot for every 1000 rows scanned. The `FEEDBACK` value applies to all tables being scanned. It cannot be set for individual tables.

FROMCHAR Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies the current character set name for `CHAR`, `VARCHAR2`, `LONG`, and `CLOB` datatypes in the database. By default, the Character Set Scanner assumes the character set for these datatypes to be the database character set.

Use this parameter to override the default database character set definition for `CHAR`, `VARCHAR2`, `LONG`, and `CLOB` data in the database.

FROMNCHAR Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies the current national database character set name for `NCHAR`, `NVARCHAR2`, and `NCLOB` datatypes in the database. By default, the Character Set Scanner assumes the character set for these datatypes to be the database national character set.

Use this parameter to override the default database character set definition for `NCHAR`, `NVARCHAR2`, and `NCLOB` data in the database.

FULL Character Set Scanner Parameter

Default value:	`N`
Range of values:	`Y` or `N`
Purpose:	Indicates whether to perform the full database scan (that is, to scan the entire database including the data dictionary). Specify `FULL=Y` to scan in full database mode.

See Also: "Scan Modes in the Character Set Scanner" on page 11-4 for more information about full database scans

HELP Character Set Scanner Parameter

Default value:	`N`
Range of values:	`Y` or `N`

Purpose:	Displays a help message with the descriptions of the Character Set Scanner parameters

See Also: "Getting Online Help for the Character Set Scanner" on page 11-7

LASTRPT Character Set Scanner Parameter

Default value:	`N`
Range of values:	`Y` or `N`
Purpose:	Indicates whether to regenerate the Character Set Scanner reports based on statistics gathered from the previous database scan

If `LASTRPT=Y` is specified, then the Character Set Scanner does not scan the database, but creates the report files using the information left by the previous database scan session instead.

If `LASTRPT=Y` is specified, then only the `USERID`, `BOUNDARIES`, and `LOG` parameters take effect.

LOG Character Set Scanner Parameter

Default value:	`scan`
Purpose:	Specifies a base file name for the following Character Set Scanner report files:

- Database Scan Summary Report file, whose extension is `.txt`
- Individual Exception Report file, whose extension is `.err`
- Screen log file, whose extension is `.out`

By default, the Character Set Scanner generates the three text files, `scan.txt`, `scan.err`, and `scan.out` in the current directory.

MAXBLOCKS Character Set Scanner Parameter

Default value:	None

Minimum value:	1000
Maximum value:	Unlimited
Purpose:	Specifies the maximum block size for each table, so that large tables can be split into smaller chunks for the Character Set Scanner to process

For example, if the `MAXBLOCKS` parameter is set to `1000`, then any tables that are greater than 1000 blocks in size will be divided into n chunks, where `n=CEIL(table block size/1000)`.

Dividing large tables into smaller pieces will be beneficial only when the number of processes set with `PROCESS` is greater than 1. If the `MAXBLOCKS` parameter is not set, then the Character Set Scanner attempts to split up large tables based on its own optimization rules.

PARFILE Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies a filename for a file that contains a list of Character Set Scanner parameters

See Also: "The Parameter File" on page 11-8

PRESERVE Character Set Scanner Parameter

Default value:	`N`
Range of values:	`Y` or `N`
Purpose:	Indicates whether to preserve the statistics gathered from the previous scan session

If `PRESERVE=Y` is specified, then the Character Set Scanner preserves all the statistics from the previous scan. It adds (if `PRESERVE=Y`) or overwrites (if `PRESERVE=N`) the new statistics for the tables being scanned in the current scan request.

PROCESS Character Set Scanner Parameter

Default value:	`1`
Minimum value:	`1`
Maximum value:	`32`
Purpose:	Specifies the number of concurrent scan processes to utilize for the database scan

SUPPRESS Character Set Scanner Parameter

Default value:	Unset (results in unlimited number of rows)
Minimum value:	`0`
Maximum value:	Unlimited
Purpose:	Specifies the maximum number of data exceptions being logged for each table

The Character Set Scanner inserts individual exceptional record information into the CSM$ERRORS table when an exception is found in a data cell. The table grows depending on the number of exceptions reported.

This parameter is used to suppress the logging of individual exception information after a specified number of exceptions are inserted for each table. For example, if SUPPRESS is set to 100, then the Character Set Scanner records a maximum of 100 exception records for each table.

See Also: "Storage Considerations" on page 11-29

TABLE Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies the names of the tables to scan

You can specify the following when you specify the name of the table:

- *`schemaname`* specifies the name of the user's schema from which to scan the table
- *`tablename`* specifies the name of the table or tables to be scanned

For example, the following command scans the `employees` and `departments` tables in the `hr` sample schema:

```
csscan system/manager TABLE=(HR.EMPLOYEES , HR.DEPARTMENTS) ...
```

TOCHAR Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies a target database character set name for the `CHAR`, `VARCHAR2`, `LONG`, and `CLOB` data

TONCHAR Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies a target database character set name for the `NCHAR`, `NVARCHAR2`, and `NCLOB` data

If you do not specify a value for `TONCHAR`, then the Character Set Scanner does not scan `NCHAR`, `NVARCHAR2`, and `NCLOB` data.

USER Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies the owner of the tables to be scanned

If the parameter `USER` is specified, then the Character Set Scanner scans all tables belonging to the user. For example, the following statement scans all tables belonging to the user `hr`:

```
csscan system/manager USER=hr ...
```

USERID Character Set Scanner Parameter

Default value:	None
Purpose:	Specifies the username and password (and optional connect string) of the user who scans the database. If you omit the password, then the Character Set Scanner prompts you for it

The following examples are all valid:

```
username/password
username/password@connect_string
username
username@connect_string
```

Examples: Character Set Scanner Sessions

The following examples show you how to use the command-line and parameter-file methods to use Full Database, User, and Table scan modes.

Example: Full Database Scan

The following example shows how to scan the full database to see the effects of migrating it to UTF8. This example assumes that the current database character set is WE8ISO8859P1 (or anything other than UTF8).

Parameter-File Method

```
% csscan system/manager parfile=param.txt
```

The `param.txt` file contains the following information:

```
full=y
tochar=utf8
array=40960
process=4
```

Command-Line Method

```
% csscan system/manager full=y tochar=utf8 array=40960 process=4

Scanner Messages
Database Scanner: Release 9.2.0.0 - Production

(c) Copyright 2001 Oracle Corporation. All rights reserved.
Connected to:
Oracle9i Enterprise Edition Release 9.2.0.0 - Production
With the Objects option
PL/SQL Release 9.2.0.0 - Production

Enumerating tables to scan...

. process 1 scanning SYSTEM.REPCAT$_RESOLUTION
. process 1 scanning SYS.AQ$_MESSAGE_TYPES
. process 1 scanning SYS.ARGUMENT$
. process 2 scanning SYS.AUD$
. process 3 scanning SYS.ATTRIBUTE$
. process 4 scanning SYS.ATTRCOL$
. process 2 scanning SYS.AUDIT_ACTIONS
. process 2 scanning SYS.BOOTSTRAP$
. process 2 scanning SYS.CCOL$
. process 2 scanning SYS.CDEF$
 :
 :
. process 3 scanning SYSTEM.REPCAT$_REPOBJECT
. process 1 scanning SYSTEM.REPCAT$_REPPROP
. process 2 scanning SYSTEM.REPCAT$_REPSCHEMA
. process 3 scanning MDSYS.MD$DIM
. process 1 scanning MDSYS.MD$DICTVER
. process 2 scanning MDSYS.MD$EXC
. process 3 scanning MDSYS.MD$LER
. process 1 scanning MDSYS.MD$PTAB
. process 2 scanning MDSYS.MD$PTS
. process 3 scanning MDSYS.MD$TAB

Creating Database Scan Summary Report...

Creating Individual Exception Report...

Scanner terminated successfully.
```

Example: User Scan

The following example shows how to scan the user tables to see the effects of migrating them to UTF8. This example assumes the current database character set is US7ASCII, but the actual data stored is in Western European WE8MSWIN1252 encoding.

Parameter-File Method

```
% csscan system/manager parfile=param.txt
```

The `param.txt` file contains the following information:

```
user=hr
fromchar=we8mswin1252
tochar=utf8
array=40960
```

```
process=1
```

Command-Line Method

```
% csscan system/manager user=hr fromchar=we8mswin1252 tochar=utf8 array=40960
process=1
```

Character Set Scanner Messages

```
Database Scanner: Release 9.2.0.0 - Production

(c) Copyright 2001 Oracle Corporation. All rights reserved.

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.0 - Production
With the Objects option
PL/SQL Release 9.2.0.0 - Production

Enumerating tables to scan...

. process 1 scanning HR.JOBS
. process 1 scanning HR.DEPARTMENTS
. process 1 scanning HR.JOB_HISTORY
. process 1 scanning HR.EMPLOYEES

Creating Database Scan Summary Report...

Creating Individual Exception Report...

Scanner terminated successfully.
```

Example: Single Table Scan

The following example shows how to scan a single table to see the effects on migrating it to WE8MSWIN1252. This example assumes the current database character set is in US7ASCII.

Parameter-File Method

```
% csscan system/manager parfile=param.txt
```

The `param.txt` file contains the following information:

```
table=employees
tochar=we8mswin1252
```

```
array=40960
process=1
supress=100
```

Command-Line Method

```
% csscan system/manager table=employees tochar=we8mswin1252 array=40960
process=1 supress=100

Scanner Messages
Database Scanner: Release 9.2.0.0 - Production

(c) Copyright 2001 Oracle Corporation. All rights reserved.

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.0 - Production
With the Objects option
PL/SQL Release 9.2.0.0 - Production

. process 1 scanning HR.EMPLOYEES

Creating Database Scan Summary Report...

Creating Individual Exception Report...

Scanner terminated successfully.
```

Character Set Scanner Reports

The Character Set Scanner generates two reports for each scan:

- Database Scan Summary Report
- Individual Exception Report

Database Scan Summary Report

A Database Scan Summary Report consists of the following sections:

- Database Parameters for the Character Set Scanner
- Database Size
- Scan Summary
- Data Dictionary Conversion Summary

- Application Data Conversion Summary
- Application Data Conversion Summary for Each Column Size Boundary
- Distribution of Convertible Data for Each Table
- Distribution of Convertible Data for Each Column
- Indexes To Be Rebuilt

The information available for each section depends on the type of scan and the parameters you select.

Database Parameters for the Character Set Scanner

This section describes the parameters selected and the type of scan you chose. The following is an example:

```
Parameter                                Value
---------------------------------------- ------------------------------
Scan type                                Full database
Scan CHAR data?                          YES
Current database character set           WE8ISO8859P1
New database character set               UTF8
Scan NCHAR data?                         NO
Array fetch buffer size                  102400
Number of processes                      4
---------------------------------------- ------------------------------
```

Database Size

This section describes the current database size. The following is an example:

```
TABLESPACE                             Total(MB)        Used(MB)        Free(MB)
----------------------------- --------------- --------------- ---------------
APPS_DATA                              1,340.000       1,331.070           8.926
CTX_DATA                                  30.000           3.145          26.852
INDEX_DATA                               140.000         132.559           7.438
RBS_DATA                                 310.000         300.434           9.563
SYSTEM_DATA                              150.000         144.969           5.027
TEMP_DATA                                160.000                         159.996
TOOLS_DATA                                35.000          22.148          12.848
USERS_DATA                               220.000         142.195          77.801
----------------------------- --------------- --------------- ---------------
Total                                  2,385.000       2,073.742         311.227
```

Scan Summary

This report indicates the feasibility of the database character set migration. There are two basic criteria that determine the feasibility of the character set migration of the database. One is the condition of the data dictionary and the other is the condition of the application data.

The Scan Summary section consists of two status lines. The scan mode and the result determines the status that is printed for the data dictionary and application data.

Table 11–1 Scan Summary for the Data Dictionary and Application Data

Possible Data Dictionary Status	Possible Application Data Status
All character-type data in the data dictionary remains the same in the new character set.	All character-type application data remains the same in the new character set.
All character-type data in the data dictionary is convertible to the new character set.	All character-type application data is convertible to the new character set.
Some character-type data in the data dictionary is not convertible to the new character set.	Some character-type application data is not convertible to the new character set.

The following is sample output:

```
All character type data in the data dictionary remains the same in the new
character set
All character type application data remains the same in the new character set
```

When all data remains the same in the new character set, it means that the data encoding of the original character set is identical to the target character set. In this case, the character set can be migrated using the ALTER DATABASE CHARACTER SET statement.

If all the data is convertible to the new character set, it means that the data can be safely migrated using the Export and Import utilities. However, the migrated data may or may not have the same encoding as the original character set.

See Also:

- "Individual Exception Report" on page 11-27 for more information about non-convertible data
- "Migrating Character Data Using the ALTER DATABASE CHARACTER SET Statement" on page 10-8
- "Migrating Character Data Using a Full Export and Import" on page 10-8

Data Dictionary Conversion Summary

This section contains the statistics on the conversion summary of the data dictionary. It reports the statistics by datatype. Table 11–2 describes the types of status that can be reported.

Table 11–2 Data Conversion Summary for the Data Dictionary

Status	Description
`Changeless`	Number of data cells that remain the same in the new character set
`Convertible`	Number of data cells that will be successfully converted to the new character set
`Exceptional`	Number of data cells that cannot be converted. If you choose to convert anyway, some characters will be lost or data will be truncated

This information is available only when a full database scan is performed. The following is sample output:

```
Datatype         Changeless      Convertible      Exceptional            Total
--------- ---------------- ---------------- ---------------- ----------------
VARCHAR2           971,300                1                0          971,301
CHAR                     7                0                0                7
LONG                60,325                0                0           60,325
CLOB
--------- ---------------- ---------------- ---------------- ----------------
Total            1,031,632                1                0        1,031,633
```

If the numbers in both the `Convertible` and `Exceptional` columns are zero, it means that all the data in the data dictionary will remain the same in the new character set.

If the numbers in the `Exceptional` column are zero and some numbers in the `Convertible` columns are not zero, it means that all data in the data dictionary is convertible to the new character set. During import, the relevant data will be converted.

If the numbers in the `Exceptional` column are not zero, it means that there is data in the data dictionary that is not convertible. Therefore, it is not feasible to migrate the current database to the new character because the export and import process cannot convert the data into the new character set. For example, you might have a table name with invalid characters or a PL/SQL procedure with a comment line that includes data that cannot be mapped to the new character set. These changes to schema objects must be corrected manually before migration to a new character set.

Application Data Conversion Summary

This section contains the statistics on conversion summary of the application data. The statistics are reported by datatype. Table 11–3 describes the types of status that can be reported.

Table 11–3 Data Conversion Summary for Application Data

Status	Description
`Changeless`	Number of data cells that remain the same in the new character set
`Convertible`	Number of data cells that will be successfully converted to the new character set
`Exceptional`	Number of data cells that cannot be converted. If you choose to convert anyway, some characters will be lost or data will be truncated

The following is sample output:

```
Datatype         Changeless      Convertible      Exceptional            Total
---------- ---------------- ---------------- ---------------- ----------------
VARCHAR2         23,213,745            1,324                0       23,215,069
CHAR                423,430                0                0          423,430
LONG                  8,624               33                0            8,657
CLOB                 58,839           11,114               28           69,981
---------- ---------------- ---------------- ---------------- ----------------
Total            23,704,638           12,471               28       23,717,137
```

Application Data Conversion Summary for Each Column Size Boundary

This section contains the conversion summary of the CHAR and VARCHAR2 application data. The statistics are reported by column size boundaries specified by the BOUNDARIES parameter. Table 11–4 describes the types of status available.

Table 11–4 Data Conversion Summary for Columns in Application Data

Status	Description
Changeless	Number of data cells that remain the same in the new character set
Convertible	Number of data cells that will be successfully converted to the new character set
Exceptional	Number of data cells that cannot be converted. If you choose to convert, some characters will be lost or data will be truncated

This information is available only when the BOUNDARIES parameter is specified.

The following is sample output:

```
Datatype                  Changeless  Convertible     Exceptional            Total
-------------------    ------------- ------------- --------------- ----------------
VARCHAR2(1..10)            1,474,825             0               0        1,474,825
VARCHAR2(11..100)          9,691,520            71               0        9,691,591
VARCHAR2(101..4000)       12,047,400         1,253               0       12,048,653
-------------------    ------------- ------------- --------------- ----------------
CHAR(1..10)                  423,413             0               0          423,413
CHAR(11..100)                     17             0               0               17
CHAR(101..4000)
-------------------    ------------- ------------- --------------- ----------------
Total                     23,637,175         1,324               0       23,638,499
```

Distribution of Convertible Data for Each Table

This report shows how Convertible and Exceptional data is distributed within the database. The statistics are reported by table. If the list contains only a few rows, it means the Convertible data is localized. If the list contains many rows, it means the Convertible data occurs throughout the database.

The following is sample output:

```
USER.TABLE                                            Convertible      Exceptional
-------------------------------------------------- ---------------- ----------------
SMG.SOURCE                                                        1                0
SMG.HELP                                                         12                0
SMG.CLOSE_LIST                                                   16                0
SMG.ATTENDEES                                                 8                0
SGT.DR_010_I1T1                                               7                0
SGT.DR_011_I1T1                                               7                0
SGT.MRK_SRV_PROFILE                                           2                0
SGT.MRK_SRV_PROFILE_TEMP                                      2                0
SGT.MRK_SRV_QUESTION                                          3                0
-------------------------------------------------- ---------------- ----------------
```

Distribution of Convertible Data for Each Column

This report shows how `Convertible` and `Exceptional` data is distributed within the database. The statistics are reported by column. The following is an example:

```
USER.TABLE|COLUMN                                     Convertible      Exceptional
-------------------------------------------------- ---------------- ----------------
SMG.SOURCE|SOURCE                                               1                0
SMG.HELP|INFO                                                  12                0
SMG.CLOSE_LIST|FNAME                                            1                0
SMG.CLOSE_LIST|LNAME                                            1                0
SMG.CLOSE_LIST|COMPANY                                          1                0
SMG.CLOSE_LIST|STREET                                           8                0
SMG.CLOSE_LIST|CITY                                             4                0
SMG.CLOSE_LIST|STATE                                            1                0
SMG.ATTENDEES|ATTENDEE_NAME                                     1                0
SMG.ATTENDEES|ADDRESS1                                          3                0
SMG.ATTENDEES|ADDRESS2                                          2                0
SMG.ATTENDEES|ADDRESS3                                          2                0
SGT.DR_010_I1T1|WORD_TEXT                                       7                0
SGT.DR_011_I1T1|WORD_TEXT                                       7                0
SGT.MRK_SRV_PROFILE|FNAME                                       1                0
SGT.MRK_SRV_PROFILE|LNAME                                       1                0
SGT.MRK_SRV_PROFILE_TEMP|FNAME                                  1                0
SGT.MRK_SRV_PROFILE_TEMP|LNAME                                  1                0
SGT.MRK_SRV_QUESTION|ANSWER                                     3                0
```

Indexes To Be Rebuilt

This generates a list of all the indexes that are affected by the database character set migration. These can be rebuilt after the data has been imported. The following is an example:

```
USER.INDEX on USER.TABLE(COLUMN)
-----------------------------------------------------------------------------
CD2000.COMPANY_IX_PID_BID_NNAME on CD2000.COMPANY(CO_NLS_NAME)
CD2000.I_MASHINE_MAINT_CONT on CD2000.MACHINE(MA_MAINT_CONT#)
CD2000.PERSON_NEWS_SABUN_CONT_CONT on
CD2000.PERSON_NEWS_SABUN_CONT(CONT_BID)
CD2000.PENEWSABUN3_PEID_CONT on CD2000.PE_NEWS_SABUN_3(CONT_BID)
```

```
PMS2000.CALLS_IX_STATUS_SUPPMGR on PMS2000.CALLS(SUPPMGR)
PMS2000.MAILQUEUE_CHK_SUB_TOM on PMS2000.MAIL_QUEUE(TO_MAIL)
PMS2000.MAILQUEUE_CHK_SUB_TOM on PMS2000.MAIL_QUEUE(SUBJECT)
PMS2000.TMP_IX_COMP on PMS2000.TMP_CHK_COMP(COMP_NAME)
---------------------------------------------------------------------------
```

Individual Exception Report

An Individual Exception Report consists of the following summaries:

- Database Scan Parameters
- Data Dictionary Individual Exceptions
- Application Data Individual Exceptions

Database Scan Parameters

This section describes the parameters and the type of scan chosen. The following is an example:

```
Parameter                                Value
---------------------------------------- ------------------------------
Scan type                                Full database
Scan CHAR data?                          YES
Current database character set           we8mswin1252
New database character set               utf8
Scan NCHAR data?                         NO
Array fetch buffer size                  102400
Number of rows to heap up for insert     10
Number of processes                      1
---------------------------------------- ------------------------------
```

Data Dictionary Individual Exceptions

This section identifies the data dictionary data that is either convertible or has exceptions. There are two types of exceptions:

- `exceed column size`
- `lossy conversion`

The following is an example of output about a data dictionary that contains convertible data:

```
User  : SYS
Table : METASTYLESHEET
Column: STYLESHEET
```

```
Type  : CLOB
Number of Exceptions         : 0
Max Post Conversion Data Size: 0

ROWID              Exception Type     Size Cell Data(first 30 bytes)
------------------ ------------------ ----- ------------------------------
AAAAHMAABAAAAs+AAA convertible
AAAAHMAABAAAAs+AAB convertible
------------------ ------------------ ----- ------------------------------
```

> **See Also:** "Application Data Individual Exceptions" on page 11-28 for more information about exceptions

Application Data Individual Exceptions

This report identifies the data that has exceptions so that this data can then be modified if necessary.

There are two types of exceptions:

- `exceed column size`

  The column size should be extended if the maximum column width has been surpassed. If not, data truncation occurs.

- `lossy conversion`

  The data must be corrected before migrating to the new character set, or else the invalid characters will be converted to a replacement character. Replacement characters are usually specified as ? or ¿ or as a character that is linguistically similar.

The following is an example of an individual exception report that illustrates some possible problems when changing the database character set from WE8ISO8859P1 to UTF8:

```
User:   HR
Table:  EMPLOYEES
Column: LAST_NAME
Type:   VARCHAR2(10)
Number of Exceptions: 2
Max Post Conversion Data Size: 11

ROWID              Exception Type     Size Cell Data(first 30 bytes)
------------------ ------------------ ----- ------------------------------
AAAA2fAAFAABJwQAAg exceed column size    11 Ährenfeldt
AAAA2fAAFAABJwQAAu lossy conversion         órâclë8™
```

```
AAAA2fAAFAABJwQAAu exceed column size    11 órâclë8™
------------------ ------------------ ----- ------------------------------
```

The values `Ährenfeldt` and `órâclë8™` exceed the column size (10 bytes) because each of the characters Ä, ó, â, and ë occupies one byte in WE8ISO8859P1 but two bytes in UTF8. The value `órâclë8™` has lossy conversion to UTF8 because the trademark sign ™ (code 153) is not a valid WE8ISO8859P1 character. It is a WE8MSWIN1252 character, which is a superset of WE8ISO8859P1.

You can view the data that has an exception by issuing a `SELECT` statement:

```
SELECT last_name FROM hr.employees
WHERE ROWID='AAAA2fAAFAABJwQAAu';
```

You can modify the data that has the exception by issuing an `UPDATE` statement:

```
UPDATE hr.employees SET last_name = 'Oracle8 TM'
WHERE ROWID='AAAA2fAAFAABJwQAAu';
```

See Also:

- "Data Truncation" on page 10-2
- "Character Set Conversion Issues" on page 10-4

Storage and Performance Considerations in the Character Set Scanner

This section describes storage and performance issues in the Character Set Scanner. It contains the following topics:

- Storage Considerations
- Performance Considerations

Storage Considerations

This section describes the size and the growth of the Character Set Scanner's system tables, and explains the approach to maintain them. There are three system tables that can increase rapidly depending on the nature of the data stored in the database.

You may want to assign a large tablespace to the user `CSMIG` by amending the `csminst.sql` script. By default, the `SYSTEM` tablespace is assigned to the user `CSMIG`.

This section includes the following topics:

- CSM$TABLES
- CSM$COLUMNS
- CSM$ERRORS

CSM$TABLES

The Character Set Scanner enumerates all tables that need to be scanned into the CSM$TABLES table.

You can look up the number of tables (to get an estimate of how large CSM$TABLES can become) in the database by issuing the following SQL statement:

```
SELECT COUNT(*) FROM DBA_TABLES;
```

CSM$COLUMNS

The Character Set Scanner stores statistical information for each column scanned into the CSM$COLUMNS table.

You can look up the number of character type columns (to get an estimate of how large CSM$COLUMNS can become) in the database by issuing the following SQL statement:

```
SELECT COUNT(*) FROM DBA_TAB_COLUMNS
WHERE DATA_TYPE IN ('CHAR', 'VARCHAR2', 'LONG', 'CLOB');
```

CSM$ERRORS

When exceptions are detected with cell data, the Character Set Scanner inserts individual exception information into the CSM$ERRORS table. This information then appears in the Individual Exception Report and facilitates identifying records to be modified if necessary.

If your database contains a lot of data that is signaled as Exceptional or Convertible (when the parameter CAPTURE=Y is set), then the CSM$ERRORS table can grow very large. You can prevent the CSM$ERRORS table from growing unnecessarily large by using the SUPPRESS parameter.

The SUPPRESS parameter applies to all tables. The Character Set Scanner suppresses inserting individual Exceptional information after the specified number of exceptions is inserted. Limiting the number of exceptions to be recorded may not be useful if the exceptions are spread over different tables.

Performance Considerations

This section describes ways to increase performance when scanning the database.

Using Multiple Scan Processes

If you plan to scan a relatively large database, for example, over 50GB, you might want to consider using multiple scan processes. This shortens the duration of database scans by using hardware resources such as CPU and memory available on the machine. A guideline for determining the number of scan processes to use is to set the number equal to the CPU_COUNT initialization parameter.

Array Fetch Buffer Size

The Character Set Scanner fetches multiple rows at a time when an array fetch is allowed. Generally, you will improve performance by letting the Character Set Scanner use a bigger array fetch buffer. Each process allocates its own fetch buffer.

Suppressing Exception and Convertible Log

The Character Set Scanner inserts individual Exceptional and Convertible (when CAPTURE=Y) information into the CSM$ERRORS table. In general, insertion into the CSM$ERRORS table is more costly than data fetching. If your database has a lot of data that is signaled as Exceptional or Convertible, then the Character Set Scanner issues many insert statements, causing performance degradation. Oracle Corporation recommends setting a limit on the number of exception rows to be recorded using the SUPRESS parameter.

Character Set Scanner Views and Messages

This section contains the following reference material:

- Character Set Scanner Views
- Character Set Scanner Error Messages

Character Set Scanner Views

The Character Set Scanner uses the following views.

CSMV$COLUMNS

This view contains statistical information about columns that were scanned.

Column	Datatype	NULL	Description
OWNER_ID	NUMBER	NOT NULL	Userid of the table owner
OWNER_NAME	VARCHAR2(30)	NOT NULL	User name of the table owner
TABLE_ID	NUMBER	NOT NULL	Object ID of the table
TABLE_NAME	VARCHAR2(30)	NOT NULL	Object name of the table
COLUMN_ID	NUMBER	NOT NULL	Column ID
COLUMN_INTID	NUMBER	NOT NULL	Internal column ID (for abstract datatypes)
COLUMN_NAME	VARCHAR2(30)	NOT NULL	Column name
COLUMN_TYPE	VARCHAR2(9)	NOT NULL	Column datatype
TOTAL_ROWS	NUMBER	NOT NULL	Number of rows in this table
NULL_ROWS	NUMBER	NOT NULL	Number of NULL data cells
CONV_ROWS	NUMBER	NOT NULL	Number of data cells that need to be converted
ERROR_ROWS	NUMBER	NOT NULL	Number of data cells that have exceptions
EXCEED_SIZE_ROWS	NUMBER	NOT NULL	Number of data cells that have exceptions
DATA_LOSS_ROWS	NUMBER	-	Number of data cells that undergo lossy conversion
MAX_POST_CONVERT_SIZE	NUMBER	-	Maximum post-conversion data size

CSMV$CONSTRAINTS

This view contains statistical information about columns that were scanned.

Column	Datatype	NULL	Description
OWNER_ID	NUMBER	NOT NULL	Userid of the constraint owner
OWNER_NAME	VARCHAR2(30)	NOT NULL	User name of the constraint owner
CONSTRAINT_ID	NUMBER	NOT NULL	Object ID of the constraint
CONSTRAINT_NAME	VARCHAR2(30)	NOT NULL	Object name of the constraint
CONSTRAINT_TYPE#	NUMBER	NOT NULL	Constraint type number
CONSTRAINT_TYPE	VARCHAR2(11)	NOT NULL	Constraint type name
TABLE_ID	NUMBER	NOT NULL	Object ID of the table
TABLE_NAME	VARCHAR2(30)	NOT NULL	Object name of the table
CONSTRAINT_RID	NUMBER	NOT NULL	Root constraint ID
CONSTRAINT_LEVEL	NUMBER	NOT NULL	Constraint level

CSMV$ERRORS

This view contains individual exception information for cell data and object definitions.

Column	Datatype	NULL	Description
OWNER_ID	NUMBER	NOT NULL	Userid of the table owner
OWNER_NAME	VARCHAR2(30)	NOT NULL	User name of the table owner
TABLE_ID	NUMBER	NOT NULL	Object ID of the table
TABLE_NAME	VARCHAR2(30)	-	Object name of the table
COLUMN_ID	NUMBER	-	Column ID
COLUMN_INTID	NUMBER	-	Internal column ID (for abstract datatypes)
COLUMN_NAME	VARCHAR2(30)	-	Column name
DATA_ROWID	VARCHAR2(1000)	-	The rowid of the data
COLUMN_TYPE	VARCHAR2(9)	-	Column datatype of object type
ERROR_TYPE	VARCHAR2(11)	-	Type of error encountered

CSMV$INDEXES

This view contains individual exception information for indexes.

Column	Datatype	NULL	Description
INDEX_OWNER_ID	NUMBER	NOT NULL	Userid of the index owner
INDEX_OWNER_NAME	VARCHAR2(30)	NOT NULL	User name of the index owner
INDEX_ID	NUMBER	NOT NULL	Object ID of the index
INDEX_NAME	VARCHAR2(30)	-	Object name of the index
INDEX_STATUS#	NUMBER	-	Status number of the index
INDEX_STATUS	VARCHAR2(8)	-	Status of the index
TABLE_OWNER_ID	NUMBER	-	Userid of the table owner
TABLE_OWNER_NAME	VARCHAR2(30)	-	User name of the table owner
TABLE_ID	NUMBER	-	Object ID of the table
TABLE_NAME	VARCHAR2(30)	-	Object name of the table
COLUMN_ID	NUMBER	-	Column ID
COLUMN_INTID	NUMBER	-	Internal column ID (for abstract datatypes)
COLUMN_NAME	VARCHAR2(30)	-	Column name

CSMV$TABLES

This view contains information about database tables to be scanned. The Character Set Scanner enumerates all tables to be scanned into this view.

Column	Datatype	NULL	Description
OWNER_ID	NUMBER	NOT NULL	Userid of the table owner
OWNER_NAME	VARCHAR2(30)	NOT NULL	User name of the table owner
TABLE_ID	NUMBER	-	Object ID of the table
TABLE_NAME	VARCHAR2(30)	-	Object name of the table
MIN_ROWID	VARCHAR2(18)	-	Minimum rowid of the split range of the table
MAX_ROWID	VARCHAR2(18)	-	Maximum rowid of the split range of the table
BLOCKS	NUMBER	-	Number of blocks in the split range
SCAN_COLUMNS	NUMBER	-	Number of columns to be scanned
SCAN_ROWS	NUMBER	-	Number of rows to be scanned
SCAN_START	VARCHAR2(8)	-	Time table scan started
SCAN_END	VARCHAR2(8)	-	Time table scan completed

Character Set Scanner Error Messages

The Character Set Scanner has the following error messages:

```
CSS-00100 failed to allocate memory size of number
An attempt was made to allocate memory with size 0 or bigger than the maximum size.
This is an internal error. Contact Oracle Customer Support.

CSS-00101 failed to release memory
An attempt was made to release memory with invalid pointer.
This is an internal error. Contact Oracle Customer Support.

CSS-00102 failed to release memory, null pointer given
An attempt was made to release memory with null pointer.
This is an internal error. Contact Oracle Customer Support.

CSS-00105 failed to parse BOUNDARIES parameter
BOUNDARIES parameter was specified in an invalid format.
Refer to the manual for the correct syntax.

CSS-00106 failed to parse SPLIT parameter
SPLIT parameter was specified in an invalid format.
Refer to the manual for the correct syntax.

CSS-00107 Character set migration utility schem not installed
```

```
CSM$VERSION table not found in the database.
Run CSMINST.SQL on the database.

CSS-00108 Character set migration utility schema not compatible
Incompatible CSM$* tables found in the database.
Run CSMINST.SQL on the database.

CSS-00110 failed to parse userid
USERID parameter was specified in an invalid format.
Refer to the manual for the correct syntax.

CSS-00111 failed to get RDBMS version
Failed to retrieve the value of the Version of the database.
This is an internal error. Contact Oracle Customer Support.

CSS-00112 database version not supported
The database version is older than release 8.0.5.0.0.
Upgrade the database to release 8.0.5.0.0 or later, then try again.

CSS-00113 user %s is not allowed to access data dictionary
The specified user cannot access the data dictionary.
Set O7_DICTIONARY_ACCESSIBILITY parameter to TRUE, or use SYS user.

CSS-00114 failed to get database character set name
Failed to retrieve value of NLS_CHARACTERSET or NLS_NCHAR_CHARACTERSET parameter from NLS_
DATABASE_PARAMETERS view.
This is an internal error. Contact Oracle Customer Support.

CSS-00115 invalid character set name %s
The specified character set is not a valid Oracle character set.
```

See Also: Appendix A, "Locale Data" for the correct character set name

```
CSS-00116 failed to reset NLS_LANG/NLS_NCHAR parameter
Failed to force NLS_LANG character set to be same as database character set.
This is an internal error. Contact Oracle Customer Support.

CSS-00117 failed to clear previous scan log
Failed to delete all rows from CSM$* tables.
This is an internal error. Contact Oracle Customer Support.

CSS-00118 failed to save command parameters
Failed to insert rows into CSM$PARAMETERS table.
This is an internal error. Contact Oracle Customer Support.

CSS-00119 failed to save scan start time
Failed to insert a row into CSM$PARAMETERS table.
This is an internal error. Contact Oracle Customer Support.
```

```
CSS-00120 failed to enumerate tables to scan
Failed to enumerate tables to scan into CSM$TABLES table.
This is an internal error. Contact Oracle Customer Support.

CSS-00121 failed to save scan complete time
Failed to insert a row into CSM$PARAMETERS table.
This is an internal error. Contact Oracle Customer Support.

CSS-00122 failed to create scan report
Failed to create database scan report.
This is an internal error. Contact Oracle Customer Support.

CSS-00123 failed to check if user %s exist
Select statement that checks if the specified user exists in the database failed.
This is an internal error. Contact Oracle Customer Support.

CSS-00124 user %s not found
The specified user does not exist in the database.
Check the user name.

CSS-00125 failed to check if table %s.%s exist
Select statement that checks if the specified table exists in the database failed.
This is an internal error. Contact Oracle Customer Support.

CSS-00126 table %s.%s not found
The specified table does not exist in the database.
Check the user name and table name.

CSS-00127 user %s does not have DBA privilege
The specified user does not have DBA privileges, which are required to scan the database.
Choose a user with DBA privileges.

CSS-00128 failed to get server version string
Failed to retrieve the version string of the database.
None.

CSS-00130 failed to initialize semaphore
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00131 failed to spawn scan process %d
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00132 failed to destroy semaphore
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00133 failed to wait semaphore
```

Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00134 failed to post semaphore
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00140 failed to scan table (tid=%d, oid=%d)
Data scan on this particular table failed.
This is an internal error. Contact Oracle Customer Support.

CSS-00141 failed to save table scan start time
Failed to update a row in the CSM$TABLES table.
This is an internal error. Contact Oracle Customer Support.

CSS-00142 failed to get table information
Failed to retrieve various information from user id and object id of the table.
This is an internal error. Contact Oracle Customer Support.

CSS-00143 failed to get column attributes
Failed to retrieve column attributes of the table.
This is an internal error. Contact Oracle Customer Support.

CSS-00144 failed to scan table %s.%s
Data scan on this particular table was not successful.
This is an internal error. Contact Oracle Customer Support.

CSS-00145 failed to save scan result for columns
Failed to insert rows into CSM$COLUMNS table.
This is an internal error. Contact Oracle Customer Support.

CSS-00146 failed to save scan result for table
Failed to update a row of CSM$TABLES table.
This is an internal error. Contact Oracle Customer Support.

CSS-00147 unexpected data truncation
Scanner allocates the exactly same size of memory as the column byte size for fetch buffer, resulting in unexpected data truncation.
This is an internal error. Contact Oracle Customer Support.

CSS-00150 failed to enumerate table
Failed to retrieve the specified table information.
This is an internal error. Contact Oracle Customer Support.

CSS-00151 failed to enumerate user tables
Failed to enumerate all tables that belong to the specified user.
This is an internal error. Contact Oracle Customer Support.

CSS-00152 failed to enumerate all tables

```
Failed to enumerate all tables in the database.
This is an internal error. Contact Oracle Customer Support.

CSS-00153 failed to enumerate character type columns
Failed to enumerate all CHAR, VARCHAR2, LONG, and CLOB columns of tables to scan.
This is an internal error. Contact Oracle Customer Support.

CSS-00154 failed to create list of tables to scan
Failed to enumerate the tables into CSM$TABLES table.
This is an internal error. Contact Oracle Customer Support.

CSS-00155 failed to split tables for scan
Failed to split the specified tables.
This is an internal error. Contact Oracle Customer Support.

CSS-00156 failed to get total number of tables to scan
Select statement that retrieves the number of tables to scan failed.
This is an internal error. Contact Oracle Customer Support.

CSS-00157 failed to retrieve list of tables to scan
Failed to read all table ids into the scanner memory.
This is an internal error. Contact Oracle Customer Support.

CSS-00158 failed to retrieve index defined on column
Select statement that retrieves index defined on the column fails.
This is an internal error. Contact Oracle Customer Support.

CSS-00160 failed to open summary report file
File open function returned error.
Check if you have create/write privilege on the disk and check if the file name specified
for the LOG parameter is valid.

CSS-00161 failed to report scan elapsed time
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00162 failed to report database size information
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00163 failed to report scan parameters
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00164 failed to report Scan summary
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00165 failed to report conversion summary
Unknown.
```

This is an internal error. Contact Oracle Customer Support.

CSS-00166 failed to report convertible data distribution
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00167 failed to open exception report file
File open function returned error.
Check if you have create/write privilege on the disk and check if the file name specified for LOG parameter is valid.

CSS-00168 failed to report individual exceptions
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00170 failed to retrieve size of tablespace %
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00171 failed to retrieve free size of tablespace %s
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00172 failed to retrieve total size of tablespace %s
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00173 failed to retrieve used size of the database
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00174 failed to retrieve free size of the database
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00175 failed to retrieve total size of the database
Unknown.
This is an internal error. Contact Oracle Customer Support.

CSS-00176 failed to enumerate user tables in bitmapped tablespace
Failed to enumerate tables in bitmapped tablespace.
This is an internal error. Contact Oracle Customer Support.

12 Customizing Locale Data

This chapter shows how to customize locale data. It includes the following topics:

Overview of the Oracle Locale Builder Utility

The Oracle Locale Builder offers an easy and efficient way to customize locale data. It provides a graphical user interface through which you can easily view, modify, and define locale-specific data. It extracts data from the text and binary definition files and presents them in a readable format so that you can process the information without worrying about the formats used in these files.

The Oracle Locale Builder handles four types of locale definitions: language, territory, character set, and linguistic sort. It also supports user-defined characters and customized linguistic rules. You can view definitions in existing text and binary definition files and make changes to them or create your own definitions.

This section contains the following topics:

- Configuring Unicode Fonts for the Oracle Locale Builder
- The Oracle Locale Builder User Interface
- Oracle Locale Builder Screens and Dialog Boxes

Configuring Unicode Fonts for the Oracle Locale Builder

The Oracle Locale Builder uses Unicode characters in many of its functions. For example, it shows the mapping of local character code points to Unicode code points.Therefore, Oracle Corporation recommends that you use a Unicode font to fully support the Oracle Locale Builder. If a character cannot be rendered with your local fonts, then it will probably be displayed as an empty box.

Font Configuration on Windows

There are many Windows `TrueType` and `OpenType` fonts that support Unicode. Oracle Corporation recommends using the Arial Unicode MS font from Microsoft, because it includes about 51,000 glyphs and supports most of the characters in Unicode 3.1.

After installing the Unicode font, add the font to the Java Runtime `font.properties` file so it can be used by the Oracle Locale Builder. The `font.properties` file is located in the `$JAVAHOME/lib` directory. For example, for the Arial Unicode MS font, add the following entries to the `font.properties` file:

```
dialog.n=Arial Unicode MS, DEFAULT_CHARSET
dialoginput.n=Arial Unicode MS, DEFAULT_CHARSET
serif.n=Arial Unicode MS, DEFAULT_CHARSET
sansserif.n=Arial Unicode MS, DEFAULT_CHARSET
```

n is the next available sequence number to assign to the Arial Unicode MS font in the font list. Java Runtime searches the font mapping list for each virtual font and use the first font available on your system.

After you edit the `font.properties` file, restart the Oracle Locale Builder.

> **See Also:** Sun's internationalization website for more information about the `font.properties` file

Font Configuration on Other Platforms

There are fewer choices of Unicode fonts for non-Windows platforms than for Windows platforms. If you cannot find a Unicode font with satisfactory character coverage, then use multiple fonts for different languages. Install each font and add the font entries into the `font.properties` file using the steps described for the Windows platform.

For example, to display Japanese characters on Sun Solaris using the font `ricoh-hg mincho`, add entries to the existing `font.properties` file in `$JAVAHOME/lib` in the `dialog`, `dialoginput`, `serif`, and `sansserif` sections. For example:

```
serif.plain.3=-ricoh-hg mincho l-medium-r-normal--*-%d-*-*-m-*-jisx0201.1976-0
```

> **See Also:** Your platform-specific documentation for more information about available fonts

The Oracle Locale Builder User Interface

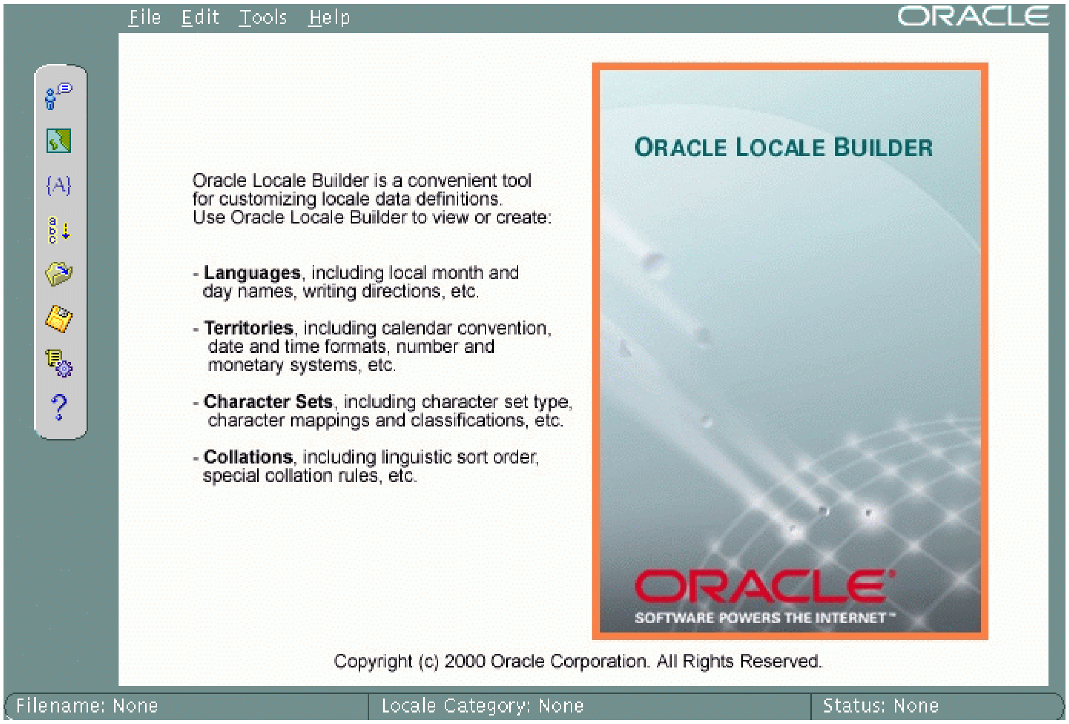
Ensure that the `ORACLE_HOME` initialization parameter is set before starting the Builder.

Start the Oracle Locale Builder by changing into the `$ORACLE_HOME/ocommon/nls/lbuilder` directory and issuing the following command:

```
% lbuilder
```

After you start the Oracle Locale Builder, the screen shown in Figure 12–1 appears.

Figure 12–1 Oracle Locale Builder Utility



Oracle Locale Builder is a convenient tool for customizing locale data definitions. Use Oracle Locale Builder to view or create:

- **Languages**, including local month and day names, writing directions, etc.
- **Territories**, including calendar convention, date and time formats, number and monetary systems, etc.
- **Character Sets**, including character set type, character mappings and classifications, etc.
- **Collations**, including linguistic sort order, special collation rules, etc.

Copyright (c) 2000 Oracle Corporation. All Rights Reserved.

Oracle Locale Builder Screens and Dialog Boxes

Before using Oracle Locale Builder for a specific task, you should become familiar with screens and dialog boxes that include the following:

- Existing Definitions Dialog Box
- Session Log Dialog Box
- Preview NLT Screen
- Open File Dialog Box

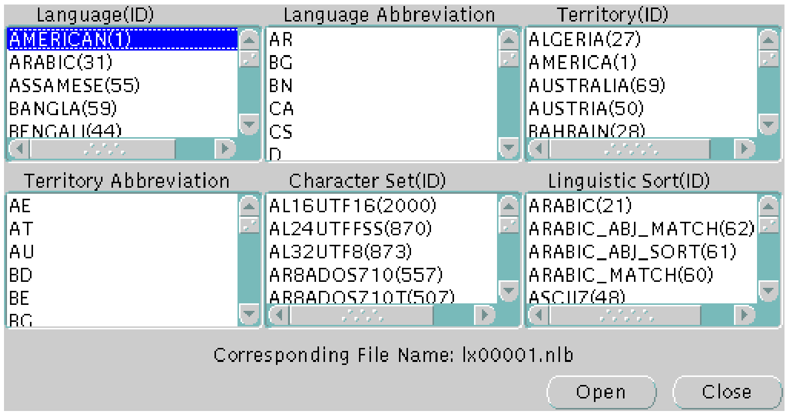
Note: Oracle Locale Builder includes online help.

Existing Definitions Dialog Box

When you choose New Language, New Territory, New Character Set, or New Linguistic Sort, the first screen you see is labelled General. Click Show Existing Definitions to see the Existing Definitions dialog box.

The Existing Definitions dialog box enables you to open locale objects by name. If you know a specific language, territory, linguistic sort (collation), or character set that you want to start with, click its displayed name. For example, you can open the `AMERICAN` language definition file as shown in Figure 12–2.

Figure 12–2 Existing Definitions Dialog Box



Language(ID)	Language Abbreviation	Territory(ID)
AMERICAN(1)	AR	ALGERIA(27)
ARABIC(31)	BG	AMERICA(1)
ASSAMESE(55)	BN	AUSTRALIA(69)
BANGLA(59)	CA	AUSTRIA(50)
BENGALI(44)	CS	BAHRAIN(28)
	D	

Territory Abbreviation	Character Set(ID)	Linguistic Sort(ID)
AE	AL16UTF16(2000)	ARABIC(21)
AT	AL24UTFFSS(870)	ARABIC_ABJ_MATCH(62)
AU	AL32UTF8(873)	ARABIC_ABJ_SORT(61)
BD	AR8ADOS710(557)	ARABIC_MATCH(60)
BE	AR8ADOS710T(507)	ASCII7(48)
BG		

Corresponding File Name: lx00001.nlb

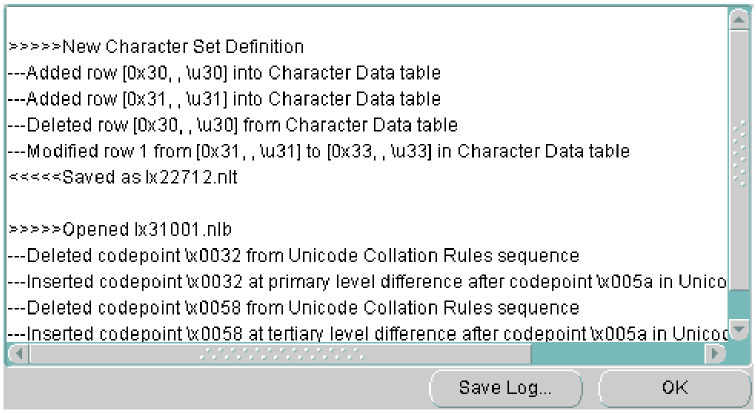
Choosing `AMERICAN` opens the `lx00001.nlb` file.

Language and territory abbreviations are for reference only and cannot be opened.

Session Log Dialog Box

In the Tools menu, choose View Log to see the Session Log dialog box. The Session Log dialog box shows what actions have been taken in the current session. The Save Log button enables you to keep a record of all changes. Figure 12–3 shows an example of a session log.

Figure 12–3 Session Log Dialog Box



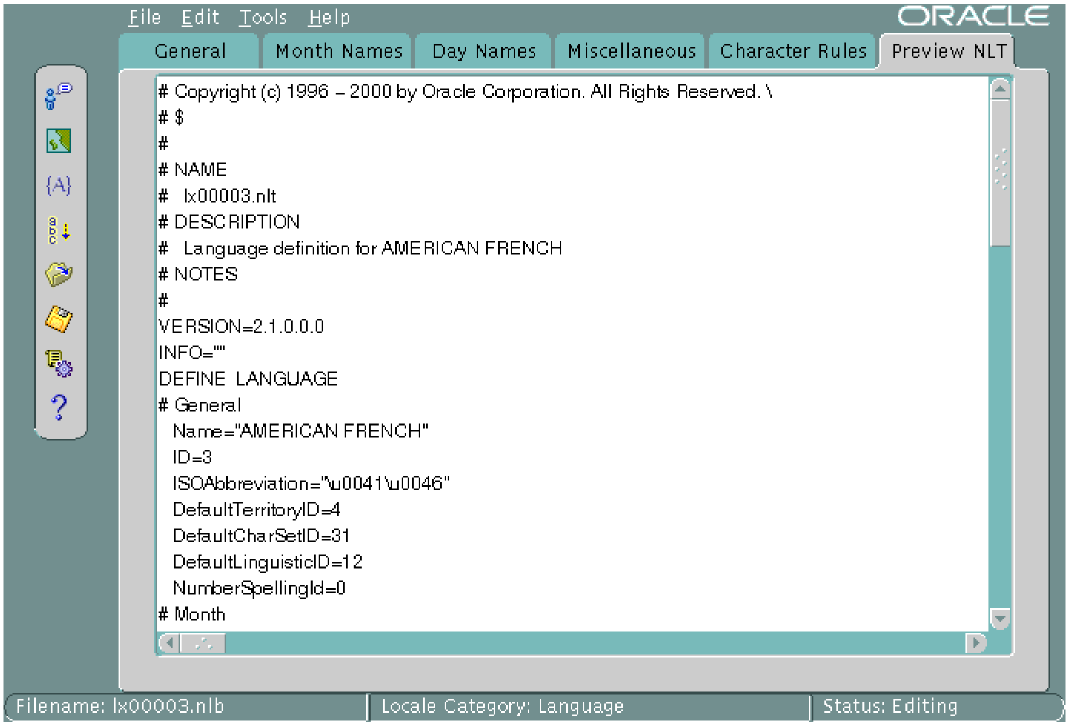
```
>>>>>New Character Set Definition
---Added row [0x30, , \u30] into Character Data table
---Added row [0x31, , \u31] into Character Data table
---Deleted row [0x30, , \u30] from Character Data table
---Modified row 1 from [0x31, , \u31] to [0x33, , \u33] in Character Data table
<<<<<Saved as lx22712.nlt

>>>>>Opened lx31001.nlb
---Deleted codepoint \x0032 from Unicode Collation Rules sequence
---Inserted codepoint \x0032 at primary level difference after codepoint \x005a in Unico
---Deleted codepoint \x0058 from Unicode Collation Rules sequence
---Inserted codepoint \x0058 at tertiary level difference after codepoint \x005a in Unicod
```

Preview NLT Screen

The NLT file is a text file with the file extension `.nlt` that shows the settings for a specific language, territory, character set, or linguistic sort. The Preview NLT screen presents a readable form of the file so that you can see whether the changes you have made look correct. You cannot modify the NLT file from the Preview NLT screen. You must use the specific elements of the Oracle Locale Builder to modify the NLT file.

Figure 12–4 shows an example of the Preview NLT screen for a user-defined language called `AMERICAN FRENCH`.

Figure 12–4 Previewing the NLT File



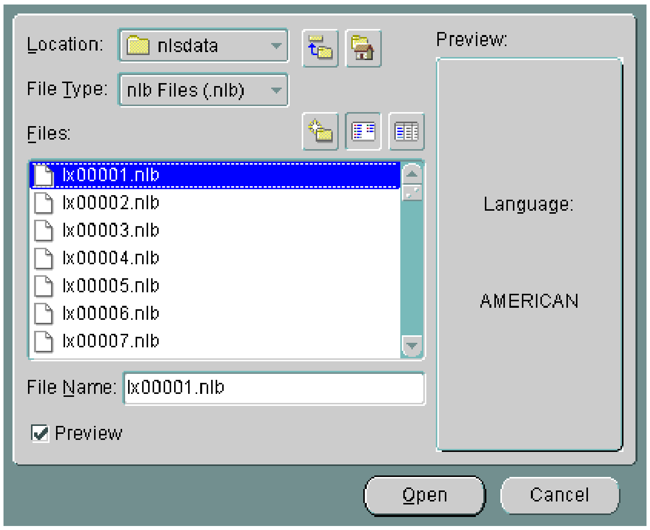
```
# Copyright (c) 1996 – 2000 by Oracle Corporation. All Rights Reserved. \
# $
#
# NAME
#   lx00003.nlt
# DESCRIPTION
#   Language definition for AMERICAN FRENCH
# NOTES
#
VERSION=2.1.0.0.0
INFO=""
DEFINE LANGUAGE
# General
  Name="AMERICAN FRENCH"
  ID=3
  ISOAbbreviation="\u0041\u0046"
  DefaultTerritoryID=4
  DefaultCharSetID=31
  DefaultLinguisticID=12
  NumberSpellingId=0
# Month
```

Filename: lx00003.nlb | Locale Category: Language | Status: Editing

Open File Dialog Box

You can see the Open File dialog box by going to the File menu, choosing Open, and choosing By File Name. Then choose the NLB file that you want to modify or use as a template. An NLB file is a binary file with the file extension `.nlb` that contains the binary equivalent of the information in the NLT file. Figure 12–5 shows the Open File dialog box with the `lx00001.nlb` file selected. The Preview panel shows that this NLB file is for the `AMERICAN` language.

Figure 12–5 Open File Dialog Box


Location: nlsdata

File Type: nlb Files (.nlb)

Files:

lx00001.nlb
lx00002.nlb
lx00003.nlb
lx00004.nlb
lx00005.nlb
lx00006.nlb
lx00007.nlb

File Name: lx00001.nlb

[✓] Preview

Preview:

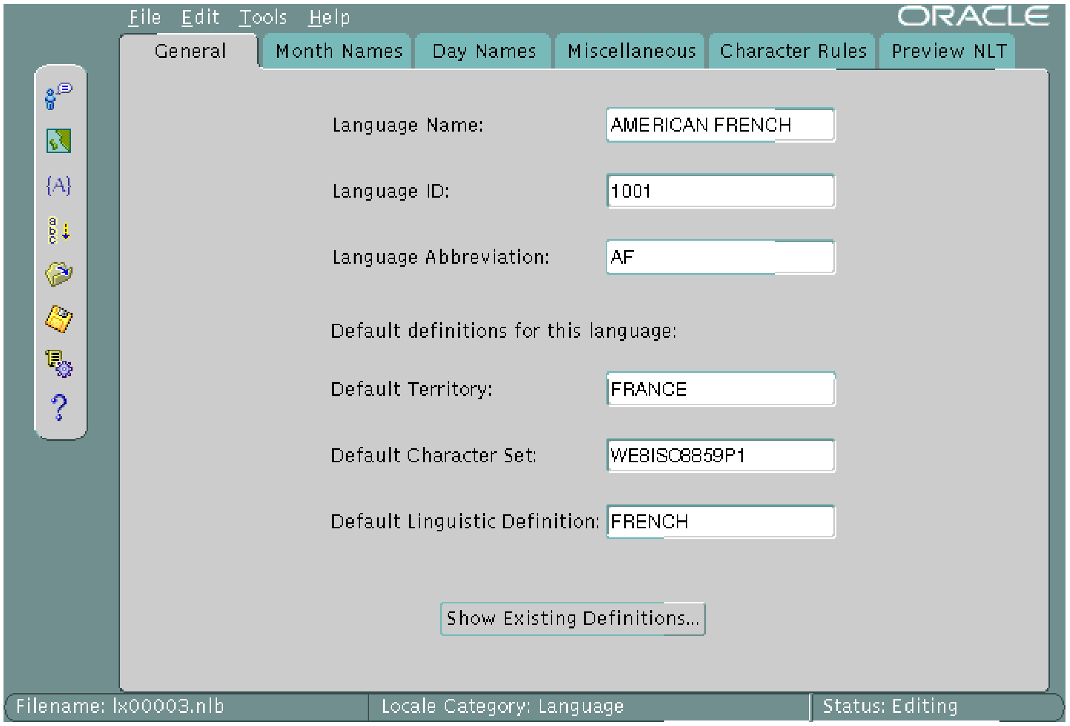
Language:

AMERICAN

Creating a New Language Definition with the Oracle Locale Builder

This section shows how to create a new language based on French. This new language is called `AMERICAN FRENCH`. First, open `FRENCH` from the Existing Definitions dialog box. Then change the language name to `AMERICAN FRENCH` and the Language Abbreviation to AF in the General dialog box. Leave the default values for the other settings. Figure 12–6 shows the resulting General dialog box.

Figure 12–6 Language General Information



Language Name: AMERICAN FRENCH

Language ID: 1001

Language Abbreviation: AF

Default definitions for this language:

Default Territory: FRANCE

Default Character Set: WE8ISO8859P1

Default Linguistic Definition: FRENCH

Filename: lx00003.nlb | Locale Category: Language | Status: Editing

The following restrictions apply when choosing names for locale objects such as languages:

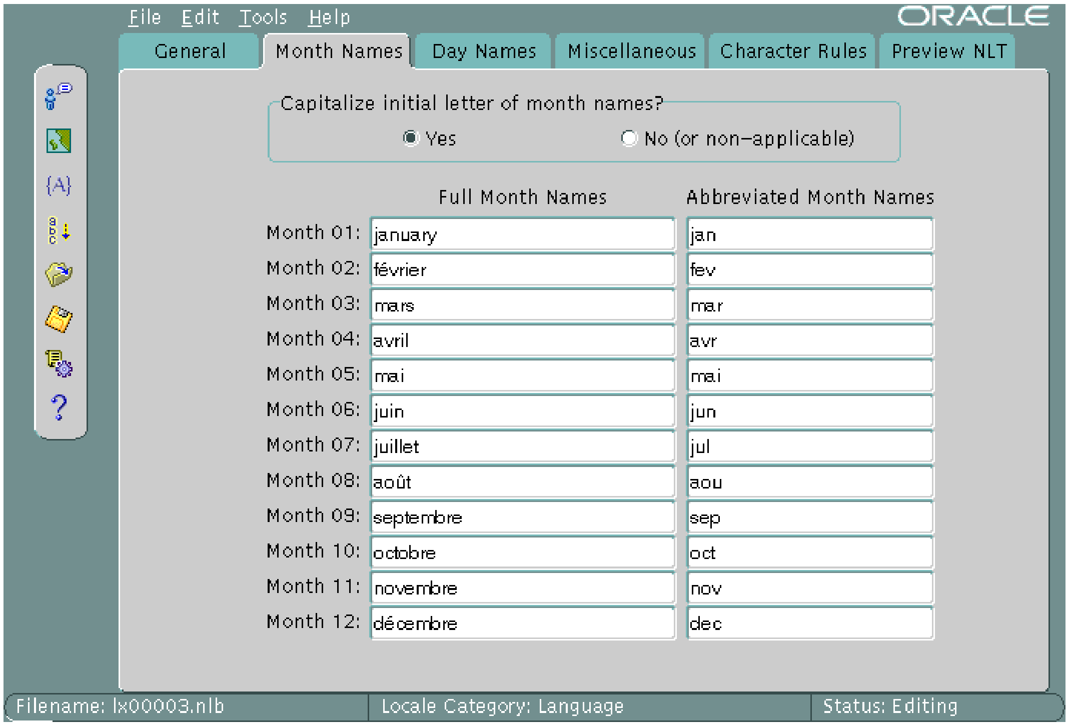
- Names must contain only ASCII characters
- Names must start with a letter
- Language, territory, and character set names cannot contain underscores

The valid range for the language ID field for a user-defined language is 1,000 to 10,000. You can accept the value provided by Oracle Locale Builder or you can specify a value within the range.

Note: Only certain ID ranges are valid values for user-defined `LANGUAGE`, `TERRITORY`, `CHARACTER SET`, `MONOLINGUAL COLLATION`, and `MULTILINGUAL COLLATION` definitions. The ranges are specified in the sections of this chapter that concern each type of user-defined locale object.

Figure 12–7 shows how to set month names using the Month Names tab.

Figure 12–7 Language Definition Month Information



Capitalize initial letter of month names?
[✓ Yes] No (or non-applicable)

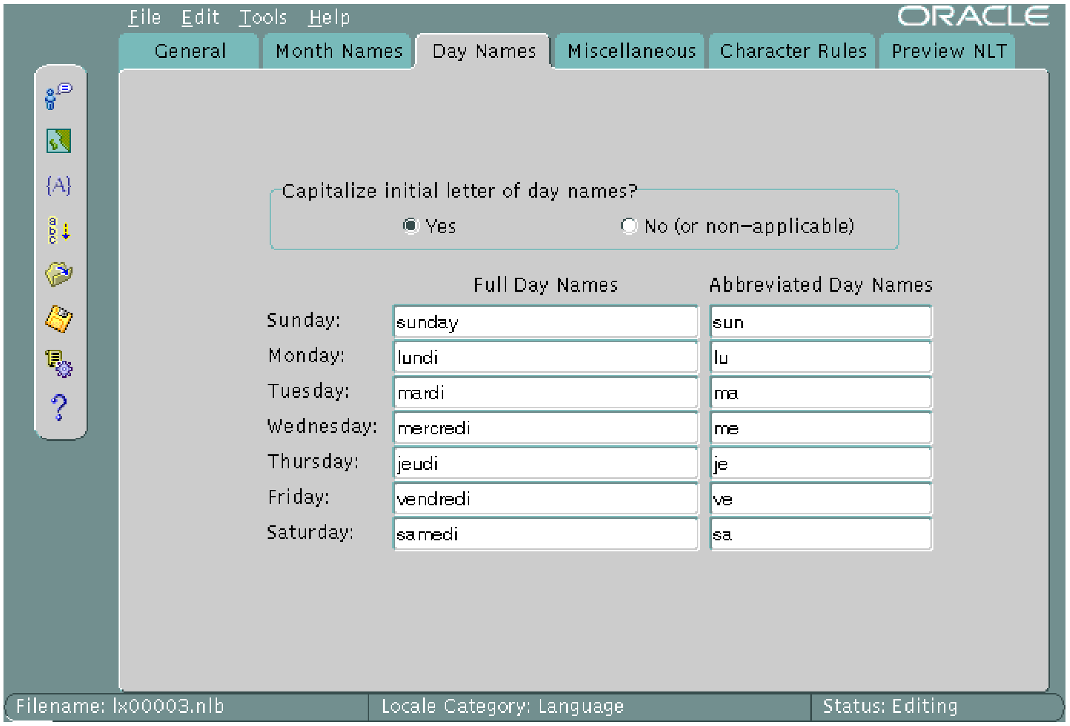
	Full Month Names	Abbreviated Month Names
Month 01:	january	jan
Month 02:	février	fev
Month 03:	mars	mar
Month 04:	avril	avr
Month 05:	mai	mai
Month 06:	juin	jun
Month 07:	juillet	jul
Month 08:	août	aou
Month 09:	septembre	sep
Month 10:	octobre	oct
Month 11:	novembre	nov
Month 12:	décembre	dec

Filename: lx00003.nlb | Locale Category: Language | Status: Editing

All names are shown as they appear in the NLT file. If you choose Yes for capitalization, the month names are capitalized in your application, but they do not appear capitalized in the Month Names screen.

Figure 12–8 shows the Day Names screen.

Figure 12–8 Language Definition Type Information



Capitalize initial letter of day names?
[✓ Yes] No (or non-applicable)

	Full Day Names	Abbreviated Day Names
Sunday:	sunday	sun
Monday:	lundi	lu
Tuesday:	mardi	ma
Wednesday:	mercredi	me
Thursday:	jeudi	je
Friday:	vendredi	ve
Saturday:	samedi	sa

Filename: lx00003.nlb | Locale Category: Language | Status: Editing

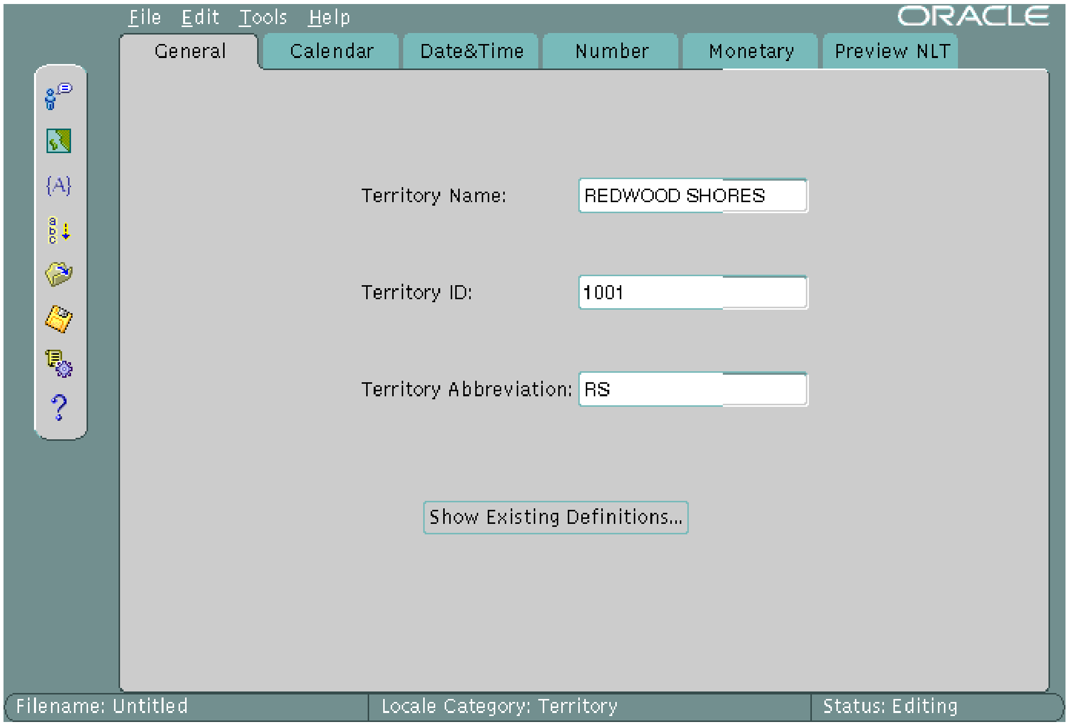
You can choose day names for your user-defined language. All names are shown as they appear in the NLT file. If you choose Yes for capitalization, the day names are capitalized in your application, but they do not appear capitalized in the Day Names screen.

Creating a New Territory Definition with the Oracle Locale Builder

This section shows how to create a new territory called `REDWOOD SHORES` and use `RS` as a territory abbreviation. The new territory is not based on an existing territory definition.

The basic tasks are to assign a territory name and choose formats for the calendar, numbers, date and time, and currency. Figure 12–9 shows the General screen with `REDWOOD SHORES` set as the Territory Name, `1001` set as the Territory ID, and `RS` set as the Territory Abbreviation.

Figure 12–9 Defining a New Territory



Territory Name: REDWOOD SHORES

Territory ID: 1001

Territory Abbreviation: RS

Filename: Untitled

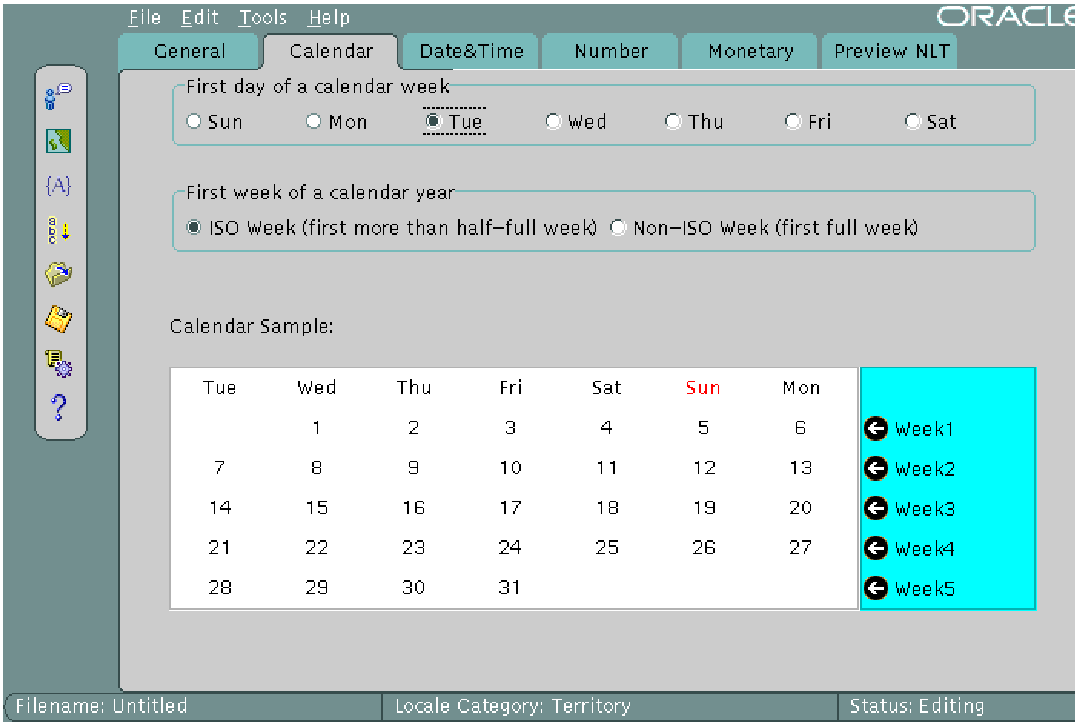
Locale Category: Territory

Status: Editing

The valid range for a territory ID for a user-defined territory is 1,000 to 10,000.

Figure 12–10 shows settings for calendar formats.

Figure 12–10 Choosing a Calendar Format



First day of a calendar week

Sun Mon [✓ Tue] Wed Thu Fri Sat

First week of a calendar year

[✓ ISO Week (first more than half-full week)] Non-ISO Week (first full week)

Calendar Sample:

Tue	Wed	Thu	Fri	Sat	Sun	Mon	
	1	2	3	4	5	6	Week1
7	8	9	10	11	12	13	Week2
14	15	16	17	18	19	20	Week3
21	22	23	24	25	26	27	Week4
28	29	30	31				Week5

Filename: Untitled | Locale Category: Territory | Status: Editing

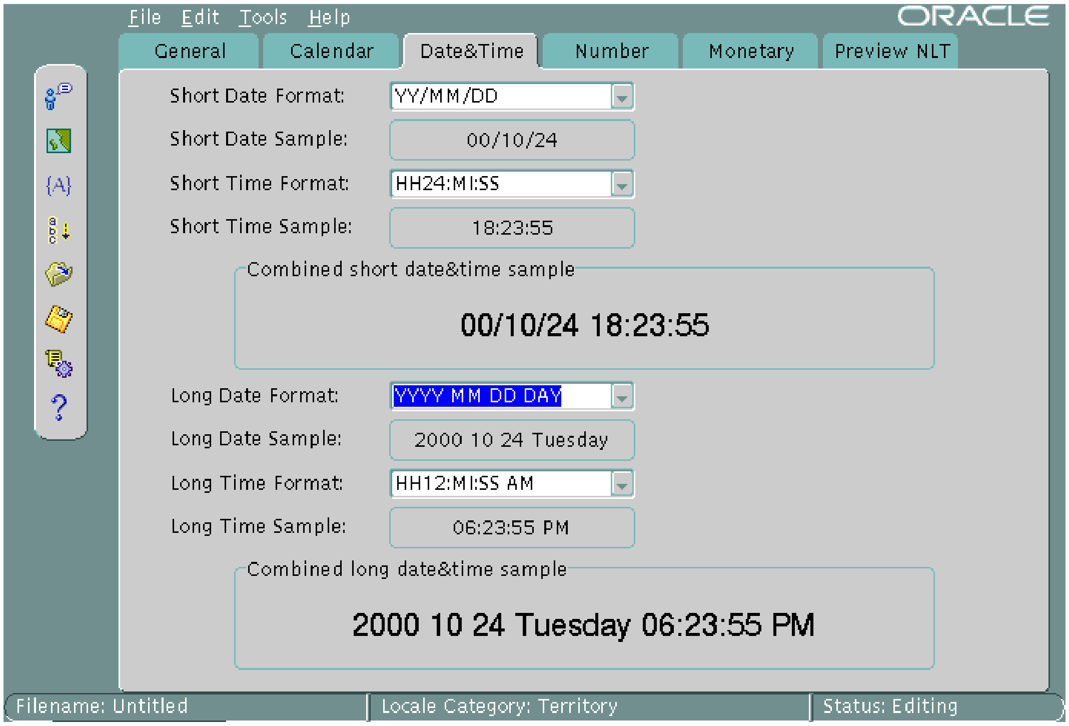
Tuesday is set as the first day of the week, and the first week of the calendar year is set as an ISO week. The screen displays a sample calendar.

See Also:

- "Calendar Formats" on page 3-26 for more information about choosing the first day of the week and the first week of the calendar year
- "Customizing Calendars with the NLS Calendar Utility" on page 12-17 for information about customizing calendars themselves

Figure 12–11 shows date and time settings.

Figure 12–11 Choosing Date and Time Formats



Short Date Format: YY/MM/DD
Short Date Sample: 00/10/24
Short Time Format: HH24:MI:SS
Short Time Sample: 18:23:55
Combined short date&time sample
00/10/24 18:23:55
Long Date Format: YYYY MM DD DAY
Long Date Sample: 2000 10 24 Tuesday
Long Time Format: HH12:MI:SS AM
Long Time Sample: 06:23:55 PM
Combined long date&time sample
2000 10 24 Tuesday 06:23:55 PM
Filename: Untitled | Locale Category: Territory | Status: Editing

Sample formats are displayed when you choose settings from the drop-down menus. In this case, the Short Date Format is set to `YY/MM/DD`. The Short Time Format is set to `HH24:MI:SS`. The Long Date Format is set to `YYYY/MM/DD DAY`. The Long Time Format is set to `HH12:MI:SS AM`.

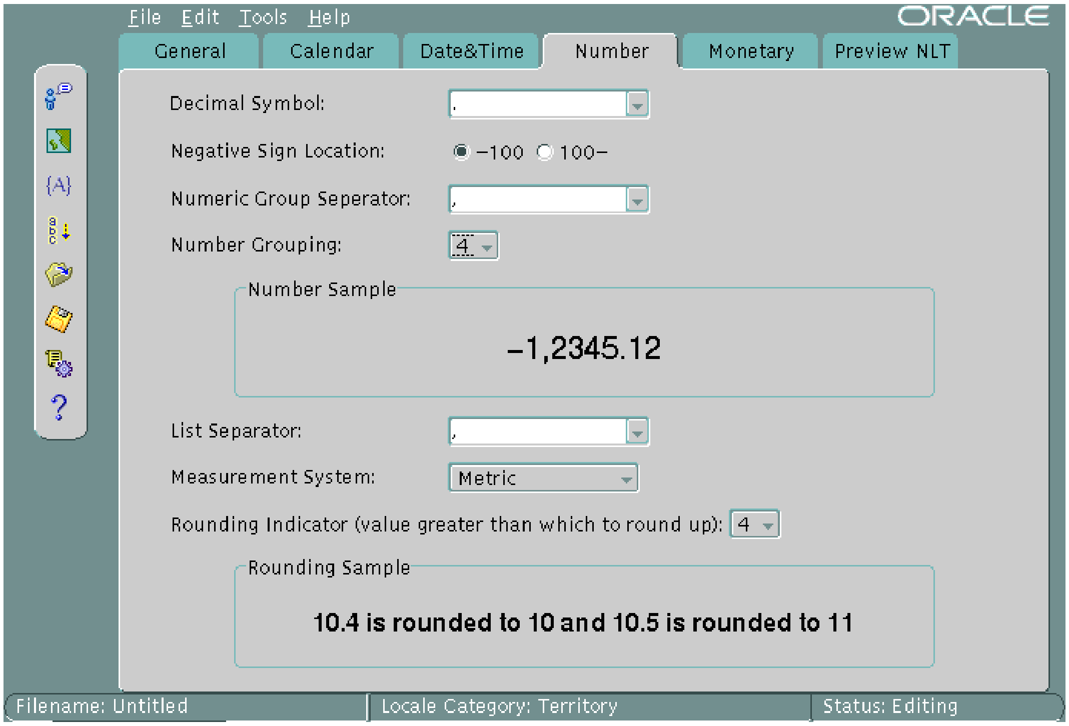
You can also enter your own formats instead of using the selection from the drop-down menus.

See Also:

- "Date Formats" on page 3-18
- "Time Formats" on page 3-21
- "Customizing Time Zone Data" on page 12-17

Figure 12–12 shows settings for number formats.

Figure 12–12 Choosing Number Formats



Decimal Symbol: .

Negative Sign Location: [✓ -100] 100-

Numeric Group Seperator: ,

Number Grouping: 4

Number Sample

-1,2345.12

List Separator: ,

Measurement System: Metric

Rounding Indicator (value greater than which to round up): 4

Rounding Sample

10.4 is rounded to 10 and 10.5 is rounded to 11

Filename: Untitled | Locale Category: Territory | Status: Editing

A period has been chosen for the Decimal Symbol. The Negative Sign Location is set to be on the left of the number. The Numeric Group Separator is a comma. The Number Grouping is set to 4 digits. The List Separator is a comma. The Measurement System is metric. The Rounding Indicator is 4.

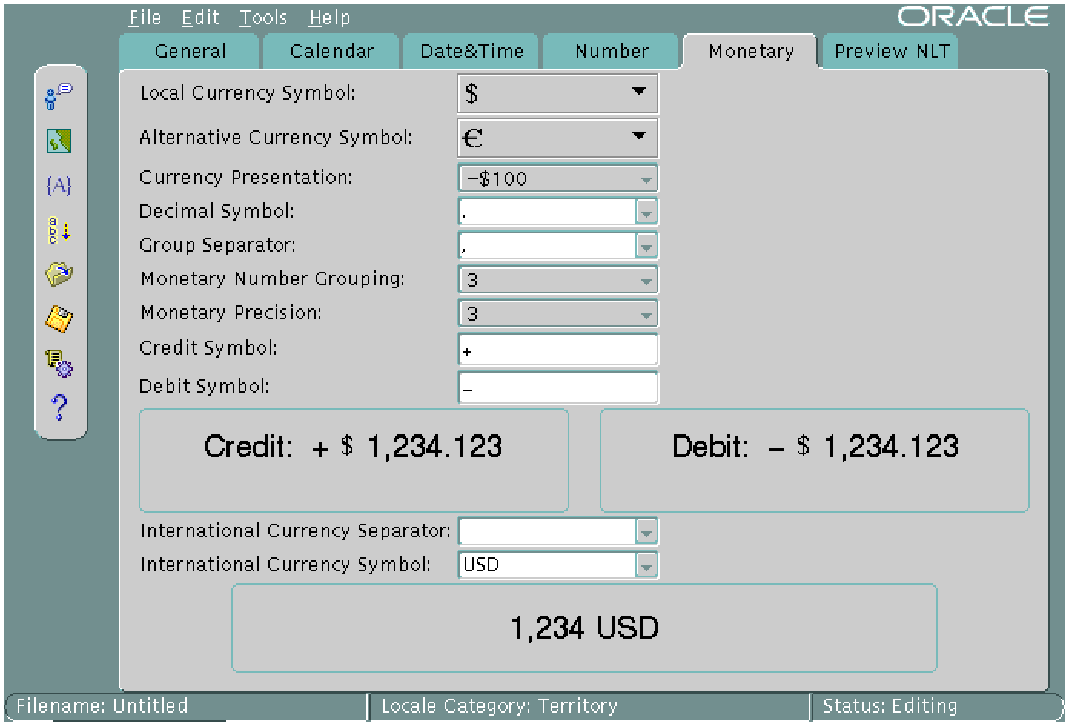
You can enter your own values instead of using the drop-down menus.

Sample formats are displayed when you choose settings from the drop-down menus.

> **See Also:** "Numeric Formats" on page 3-30

Figure 12–13 shows settings for currency formats in the Monetary dialog box.

Figure 12–13 Choosing Currency Formats



Local Currency Symbol: $
Alternative Currency Symbol: €
Currency Presentation: -$100
Decimal Symbol: .
Group Separator: ,
Monetary Number Grouping: 3
Monetary Precision: 3
Credit Symbol: +
Debit Symbol: -

Credit: + $ 1,234.123

Debit: – $ 1,234.123

International Currency Separator:
International Currency Symbol: USD

1,234 USD

Filename: Untitled | Locale Category: Territory | Status: Editing

The Local Currency Symbol is set to $. The Alternative Currency Symbol is the Euro symbol. The Currency Presentation shows one of several possible sequences of the local currency symbol, the debit symbol, and the number. The Decimal Symbol is the period. The Group Separator is the comma. The Monetary Number Grouping is 3. The Monetary Precision, or number of digits after the decimal symbol, is 3. The Credit Symbol is +. The Debit Symbol is -. The International Currency Separator is a blank space, so it is not visible in the screen. The International Currency Symbol (ISO currency symbol) is USD. Sample currency formats are displayed, based on the values you have selected.

You can enter your own values instead of using the drop-down menus.

See Also: "Currency Formats" on page 3-32

The rest of this section contains the following topics:

- Customizing Time Zone Data
- Customizing Calendars with the NLS Calendar Utility

Customizing Time Zone Data

The time zone files contain the valid time zone names. The following information is included for each time zone:

- Offset from Coordinated Universal Time (UTC)
- Transition times for daylight savings time
- Abbreviations for standard time and daylight savings time. The abbreviations are used with the time zone names.

Two time zone files are included in the Oracle home directory. The default file is `oracore/zoneinfo/timezone.dat`. It contains the most commonly used time zones. A larger set of time zones is included in `oracore/zoneinfo/timezlrg.dat`. Unless you need the larger set of time zones, use the default time zone file because database performance is better.

To use the larger time zone file, complete the following tasks:

1. Shut down the database.
2. Set the `ORA_TZFILE` environment variable to the full path name of the `timezlrg.dat` file.
3. Restart the database.

After you have used the `timezlrg.dat` file, you must continue to use it unless you are sure that none of the additional time zones are used for data that is stored in the database. Also, all databases that share information must use the same time zone file.

To view the time zone names, enter the following statement:

```
SQL> SELECT * FROM V$TIMEZONE_NAMES;
```

Customizing Calendars with the NLS Calendar Utility

Oracle supports several calendars. All of them are defined with data derived from Oracle's globalization support, but some of them may require the addition of ruler eras or deviation days in the future. To add this information without waiting for a new release of the Oracle database server, you can use an external file that is automatically loaded when the calendar functions are executed.

Calendar data is first defined in a text file. The text definition file must be converted into binary format. You can use the NLS Calendar Utility (`lxegen`) to convert the text definition file into binary format.

The name of the text definition file and its location are hard-coded and depend on the platform. On UNIX platforms, the file name is `lxecal.nlt`. It is located in the `$ORACLE_HOME/ocommon/nls` directory. A sample text definition file is included in the directory.

The `lxegen` utility produces a binary file from the text definition file. The name of the binary file is also hard-coded and depends on the platform. On UNIX platforms, the name of the binary file is `lxecal.nlb`. The binary file is generated in the same directory as the text file and overwrites an existing binary file.

After the binary file has been generated, it is automatically loaded during system initialization. Do not move or rename the file.

Invoke the calendar utility from the command line as follows:

```
% lxegen
```

See Also:

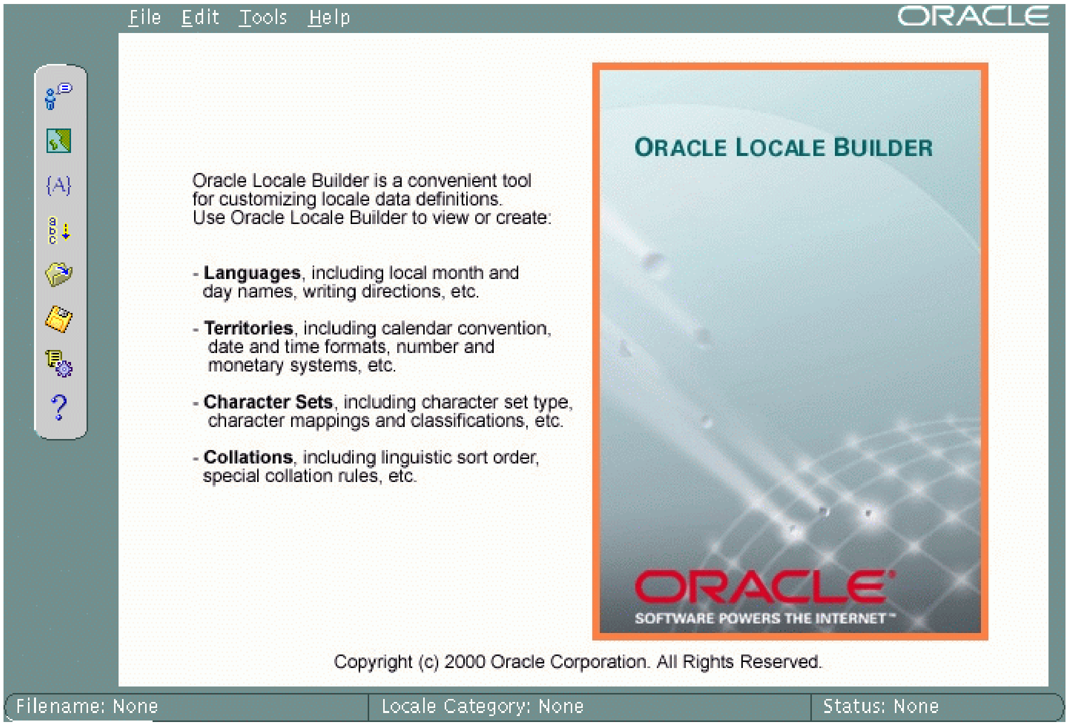
- Platform-specific documentation for the location of the files on your system
- "Calendar Systems" on page A-25

Displaying a Code Chart with the Oracle Locale Builder

You can display and print the code charts of character sets with the Oracle Locale Builder.

Figure 12–14 shows the opening screen for Oracle Locale Builder.

Figure 12–14 Opening Screen for Oracle Locale Builder



Oracle Locale Builder is a convenient tool for customizing locale data definitions. Use Oracle Locale Builder to view or create:

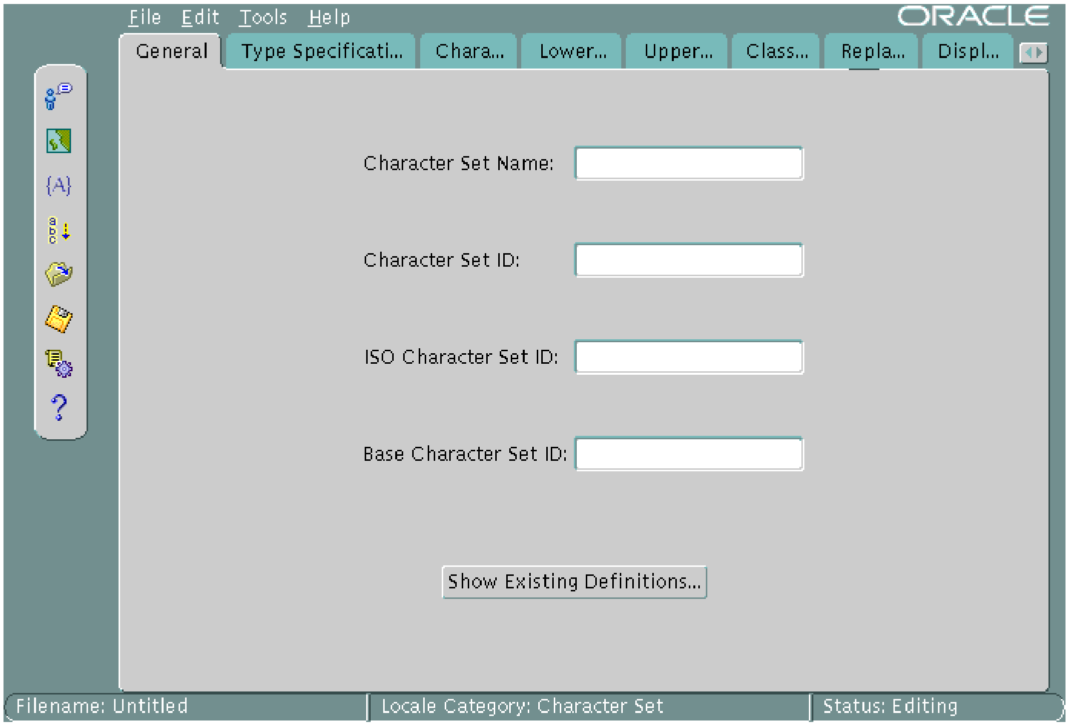
- **Languages**, including local month and day names, writing directions, etc.
- **Territories**, including calendar convention, date and time formats, number and monetary systems, etc.
- **Character Sets**, including character set type, character mappings and classifications, etc.
- **Collations**, including linguistic sort order, special collation rules, etc.

Copyright (c) 2000 Oracle Corporation. All Rights Reserved.

Filename: None | Locale Category: None | Status: None

In the File menu, choose New. In the New menu, choose Character Set. Figure 12–15 shows the resulting screen.

Figure 12–15 General Character Set Screen



Character Set Name:

Character Set ID:

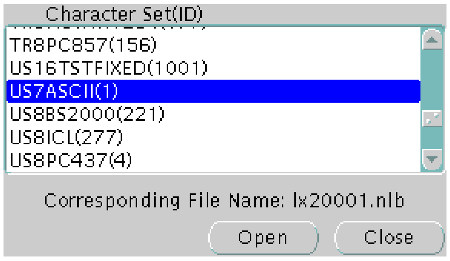
ISO Character Set ID:

Base Character Set ID:

Filename: Untitled | Locale Category: Character Set | Status: Editing

Click Show Existing Definitions. Highlight the character set you wish to display. Figure 12–16 shows the Existing Definitions dialog box with US7ASCII highlighted.

Figure 12–16 Choosing US7ASCII in the Existing Definitions Dialog Box

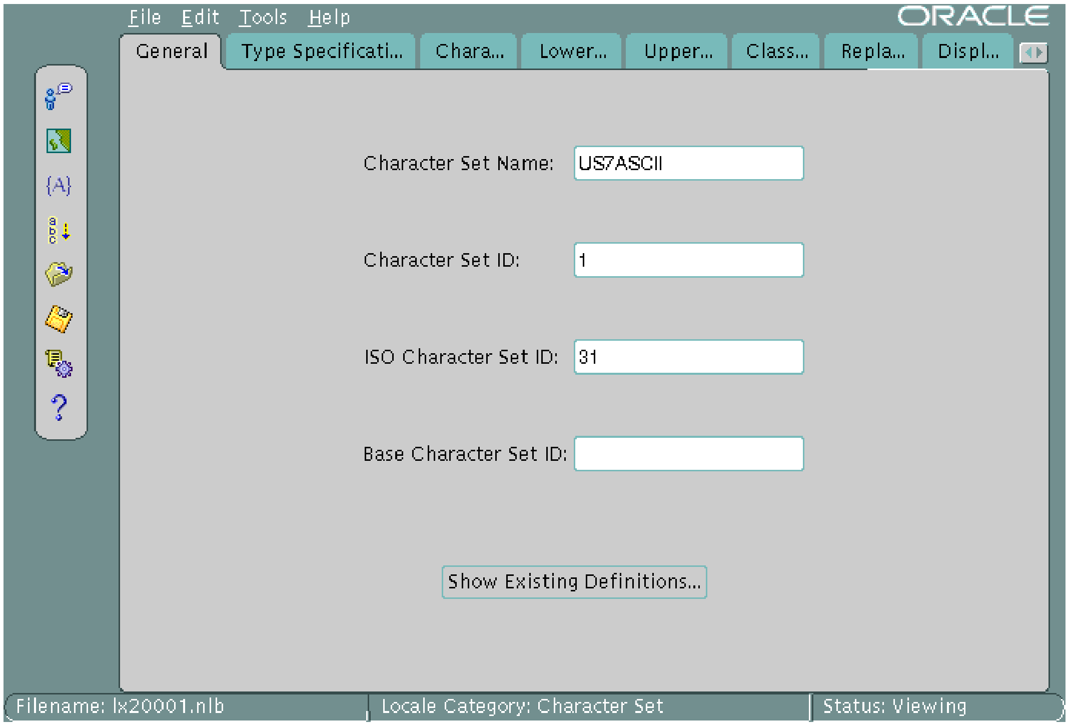


Character Set(ID)
TR8PC857(156)
US16TSTFIXED(1001)
US7ASCII(1)
US8BS2000(221)
US8ICL(277)
US8PC437(4)

Corresponding File Name: lx20001.nlb

Click Open to choose the character set. Figure 12–17 shows the General screen when US7ASCII has been chosen.

Figure 12–17 General Screen When US7ASCII Has Been Loaded



Character Set Name: US7ASCII

Character Set ID: 1

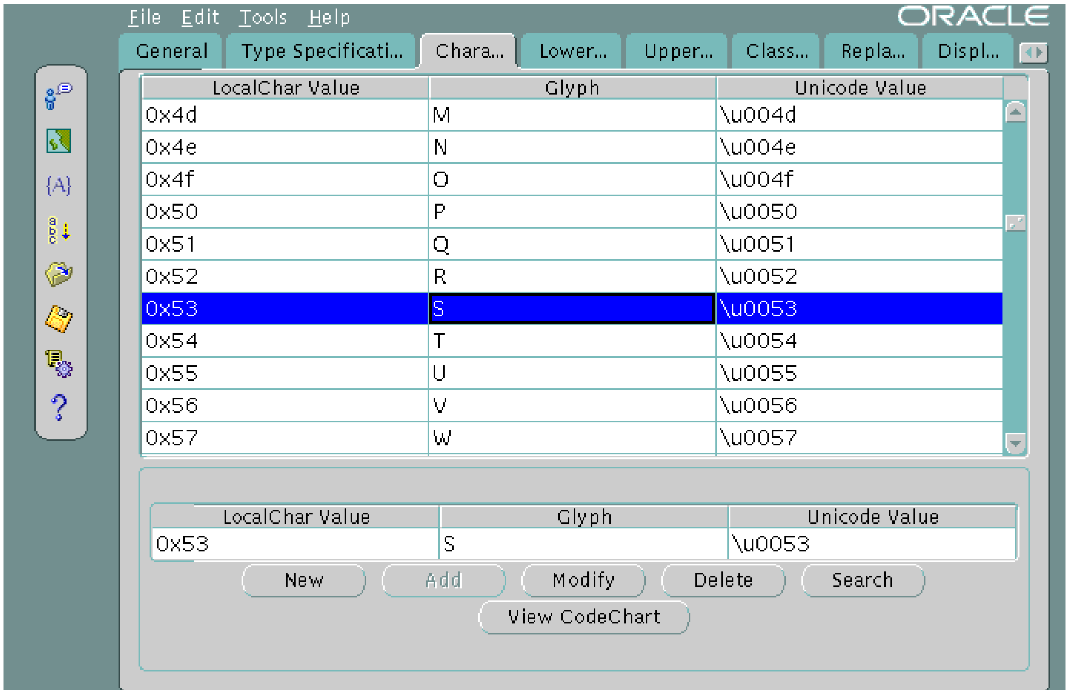
ISO Character Set ID: 31

Base Character Set ID:

Filename: lx20001.nlb | Locale Category: Character Set | Status: Viewing

Click the Character Data Mapping tab. Figure 12–18 shows the Character Data Mapping screen for US7ASCII.

Figure 12–18 Character Data Mapping for US7ASCII

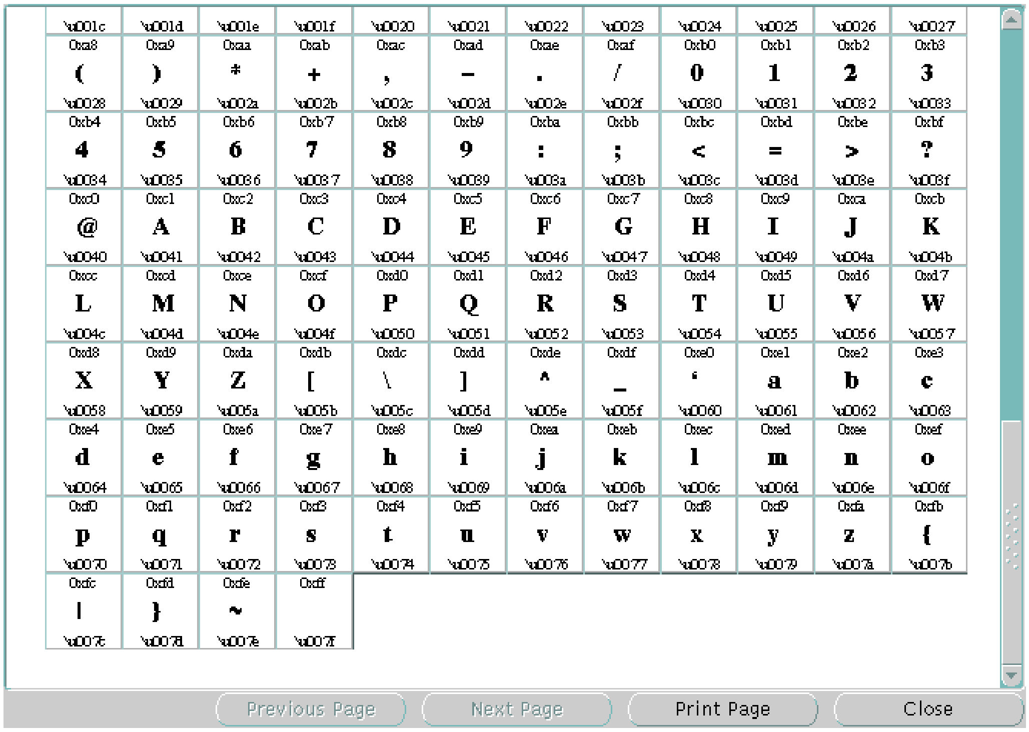


LocalChar Value	Glyph	Unicode Value
0x4d	M	\u004d
0x4e	N	\u004e
0x4f	O	\u004f
0x50	P	\u0050
0x51	Q	\u0051
0x52	R	\u0052
0x53	S	\u0053
0x54	T	\u0054
0x55	U	\u0055
0x56	V	\u0056
0x57	W	\u0057

LocalChar Value	Glyph	Unicode Value
0x53	S	\u0053

Click View CodeChart. Figure 12–19 shows the code chart for US7ASCII.

Figure 12–19 US7ASCII Code Chart



\u001c	\u001d	\u001e	\u001f	\u0020	\u0021	\u0022	\u0023	\u0024	\u0025	\u0026	\u0027
0xa8 (\u0028	0xa9) \u0029	0xaa * \u002a	0xab + \u002b	0xac , \u002c	0xad - \u002d	0xae . \u002e	0xaf / \u002f	0xb0 0 \u0030	0xb1 1 \u0031	0xb2 2 \u0032	0xb3 3 \u0033
0xb4 4 \u0034	0xb5 5 \u0035	0xb6 6 \u0036	0xb7 7 \u0037	0xb8 8 \u0038	0xb9 9 \u0039	0xba : \u003a	0xbb ; \u003b	0xbc < \u003c	0xbd = \u003d	0xbe > \u003e	0xbf ? \u003f
0xc0 @ \u0040	0xc1 A \u0041	0xc2 B \u0042	0xc3 C \u0043	0xc4 D \u0044	0xc5 E \u0045	0xc6 F \u0046	0xc7 G \u0047	0xc8 H \u0048	0xc9 I \u0049	0xca J \u004a	0xcb K \u004b
0xcc L \u004c	0xcd M \u004d	0xce N \u004e	0xcf O \u004f	0xd0 P \u0050	0xd1 Q \u0051	0xd2 R \u0052	0xd3 S \u0053	0xd4 T \u0054	0xd5 U \u0055	0xd6 V \u0056	0xd7 W \u0057
0xd8 X \u0058	0xd9 Y \u0059	0xda Z \u005a	0xdb [\u005b	0xdc \\ \u005c	0xdd] \u005d	0xde ^ \u005e	0xdf _ \u005f	0xe0 ` \u0060	0xe1 a \u0061	0xe2 b \u0062	0xe3 c \u0063
0xe4 d \u0064	0xe5 e \u0065	0xe6 f \u0066	0xe7 g \u0067	0xe8 h \u0068	0xe9 i \u0069	0xea j \u006a	0xeb k \u006b	0xec l \u006c	0xed m \u006d	0xee n \u006e	0xef o \u006f
0xf0 p \u0070	0xf1 q \u0071	0xf2 r \u0072	0xf3 s \u0073	0xf4 t \u0074	0xf5 u \u0075	0xf6 v \u0076	0xf7 w \u0077	0xf8 x \u0078	0xf9 y \u0079	0xfa z \u007a	0xfb { \u007b
0xfc | \u007c	0xfd } \u007d	0xfe ~ \u007e	0xff \u007f								

It shows the encoded value of each character in the local character set, the glyph associated with each character, and the Unicode value of each character in the local character set.

If you want to print the code chart, then click Print Page.

Creating a New Character Set Definition with the Oracle Locale Builder

You can customize a character set to meet specific user needs. In Oracle9*i*, you can extend an existing encoded character set definition. User-defined characters are often used to encode special characters that represent the following:

- Proper names
- Historical Han characters that are not defined in an existing character set standard

- Vendor-specific characters
- New symbols or characters that you define

This section describes how Oracle supports user-defined characters. It includes the following topics:

- Character Sets with User-Defined Characters
- Oracle Character Set Conversion Architecture
- Unicode 3.1 Private Use Area
- User-Defined Character Cross-References Between Character Sets
- Guidelines for Creating a New Character Set from an Existing Character Set
- Example: Creating a New Character Set Definition with the Oracle Locale Builder
- Supporting User-Defined Characters in Java

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 code points for user-defined characters. For example, Japanese Shift-JIS preserves 1880 code points for user-defined characters. They are shown in Table 12–1.

Table 12–1 Shift JIS User-Defined Character Ranges

Japanese Shift JIS User-Defined Character Range	Number of Code Points
F040-F07E, F080-F0FC	188
F140-F17E, F180-F1FC	188
F240-F27E, F280-F2FC	188
F340-F37E, F380-F3FC	188
F440-F47E, F480-F4FC	188
F540-F57E, F580-F5FC	188
FF640-F67E, F680-F6FC	188
F740-F77E, F780-F7FC	188
F840-F87E, F880-F8FC	188

Table 12–1 Shift JIS User-Defined Character Ranges (Cont.)

Japanese Shift JIS User-Defined Character Range	Number of Code Points
F940-F97E, F980-F9FC	188

The Oracle character sets listed in Table 12–2 contain predefined ranges that support user-defined characters.

Table 12–2 Oracle Character Sets with User-Defined Character Ranges

Character Set Name	Number of Code Points Available for User-Defined Characters
JA16DBCS	4370
JA16EBCDIC930	4370
JA16SJIS	1880
JA16SJISYEN	1880
KO16DBCS	1880
KO16MSWIN949	1880
ZHS16DBCS	1880
ZHS16GBK	2149
ZHT16DBCS	6204
ZHT16MSWIN950	6217

Oracle Character Set Conversion Architecture

The code point value that represents a particular character can vary among different character sets. A Japanese kanji character is shown in Figure 12–20.

Figure 12–20 Japanese Kanji Character

亜

The following table shows how the character is encoded in different character sets.

Unicode Encoding	JA16SJIS Encoding	JA16EUC Encoding	JA16DBCS Encoding
4E9C	889F	B0A1	4867

In Oracle, all character sets are defined in terms of Unicode 3.1 code points. That is, each character is defined as a Unicode 3.1 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 inFigure 12–20 has the code point value 889F when it is entered from the JA16SJIS client. It is internally converted to Unicode (with code point value 4E9C) and then converted to JA16EU (code point value B0A1).

Unicode 3.1 Private Use Area

Unicode 3.1 reserves the range E000-F8FF for the Private Use Area (PUA). The PUA is intended for private use character definition by end users or vendors.

User-defined characters can be converted between two Oracle character sets by using Unicode 3.1 PUA as the intermediate form, the same as standard characters.

User-Defined Character Cross-References Between Character Sets

User-defined character cross-references between Japanese character sets, Korean character sets, Simplified Chinese character sets and Traditional Chinese character sets are contained in the following distribution sets:

```
${ORACLE_HOME}/ocommon/nls/demo/udc_ja.txt
${ORACLE_HOME}/ocommon/nls/demo/udc_ko.txt
${ORACLE_HOME}/ocommon/nls/demo/udc_zhs.txt
${ORACLE_HOME}/ocommon/nls/demo/udc_zht.txt
```

These cross-references are useful when registering user-defined characters across operating systems. For example, when registering a new user-defined character on both a Japanese Shift-JIS operating system and a Japanese IBM Host operating system, you may want to use F040 on the Shift-JIS operating system and 6941 on IBM Host operating system for the new user-defined character so that Oracle can convert correctly between JA16SJIS and JA16DBCS. You can find out that both Shift-JIS UDC value F040 and IBM Host UDC value 6941 are mapped to the same Unicode PUA value E000 in the user-defined character cross-reference.

See Also: Appendix B, "Unicode Character Code Assignments"

Guidelines for Creating a New Character Set from an Existing Character Set

By default, the Oracle Locale Builder generates the next available character set name for you. You can also generate your own character set name. Use the following format for naming character set definition NLT files:

```
lx2dddd.nlt
```

dddd is the 4-digit Character Set ID in hex.

When you modify a character set, observe the following guidelines:

- Do not remap existing characters.
- All character mappings must be unique.
- New characters should be mapped into the Unicode private use range e000 to f4ff. (Note that the actual Unicode 3.1 private use range is e000-f8ff. However, Oracle reserves f500-f8ff for its own private use.)
- No line in the character set definition file can be longer than 80 characters.

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

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

For example, if a company such as Sun Microsystems adds user-defined characters to the JA16EUC character set, the following character set name is appropriate:

```
JA16EUCSUNWEXT1
```

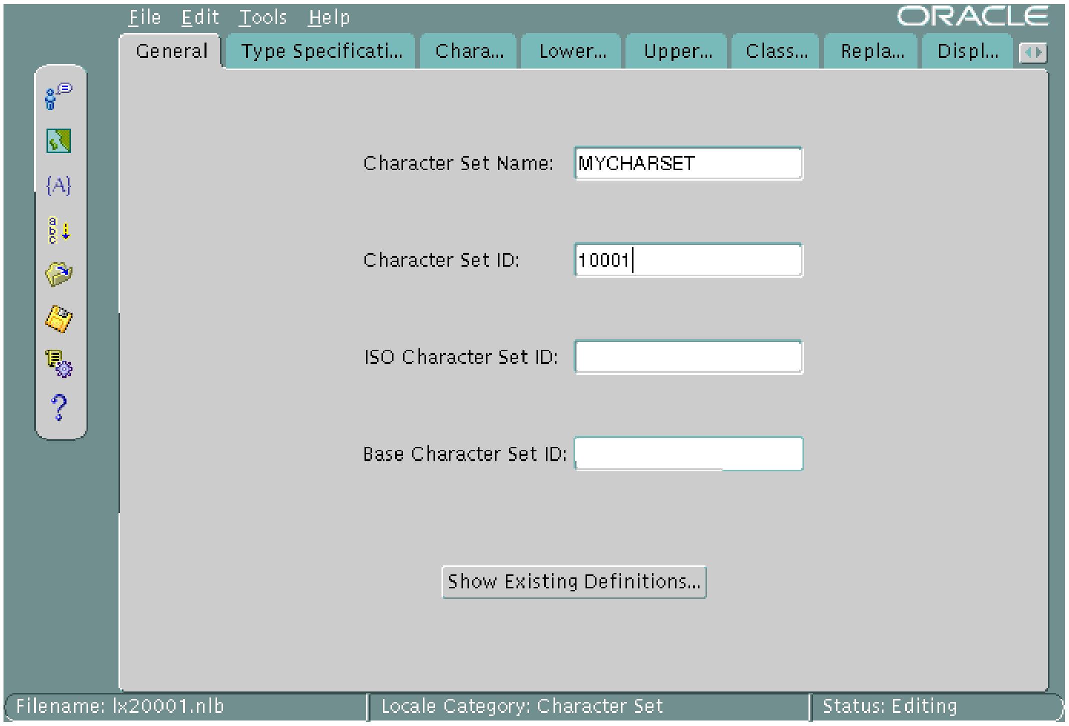
The character set name contains the following parts:

- `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 character set is an extension to the JA16EUC character set
- `1` specifies the version

Example: Creating a New Character Set Definition with the Oracle Locale Builder

This section shows how to create a new character set called `MYCHARSET` with `10001` for its Character Set ID. The example starts with the US7ASCII character set and adds 10 Chinese characters. Figure 12–21 shows the General screen.

Figure 12–21 Character Set General Information



Character Set Name: MYCHARSET

Character Set ID: 10001

ISO Character Set ID:

Base Character Set ID:

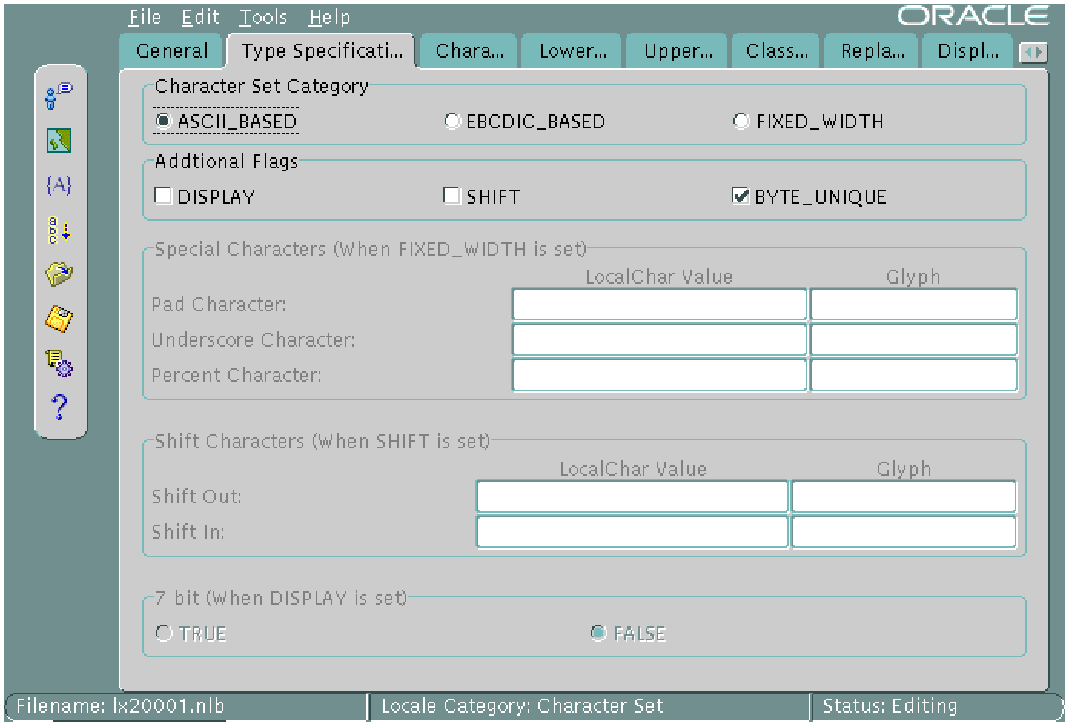
Filename: lx20001.nlb | Locale Category: Character Set | Status: Editing

Click Show Existing Definitions and choose the US7ASCII character set from the Existing Definitions dialog box.

The ISO Character Set ID and Base Character Set ID fields are optional. The Base Character Set ID is used for inheriting values so that the properties of the base character set are used as a template. The Character Set ID is automatically generated, but you can override it. The valid range for a user-defined character set ID is 10,000 to 20,000. The ISO Character Set ID field remains blank for user-defined character sets.

Figure 12–22 shows the Type Specification screen.

Figure 12–22 Character Set Type Specification



Character Set Category
[● ASCII_BASED] ○ EBCDIC_BASED ○ FIXED_WIDTH

Addtional Flags
☐ DISPLAY ☐ SHIFT [✓ BYTE_UNIQUE]

Special Characters (When FIXED_WIDTH is set)

	LocalChar Value	Glyph
Pad Character:		
Underscore Character:		
Percent Character:		

Shift Characters (When SHIFT is set)

	LocalChar Value	Glyph
Shift Out:		
Shift In:		

7 bit (When DISPLAY is set)
○ TRUE [● FALSE]

Filename: lx20001.nlb | Locale Category: Character Set | Status: Editing

The Character Set Category is `ASCII_BASED`. The `BYTE_UNIQUE` flag is checked.

When you have chosen an existing character set, the fields for the Type Specification screen should already be set to appropriate values. You should keep these values unless you have a specific reason for changing them. If you need to change the settings, use the following guidelines:

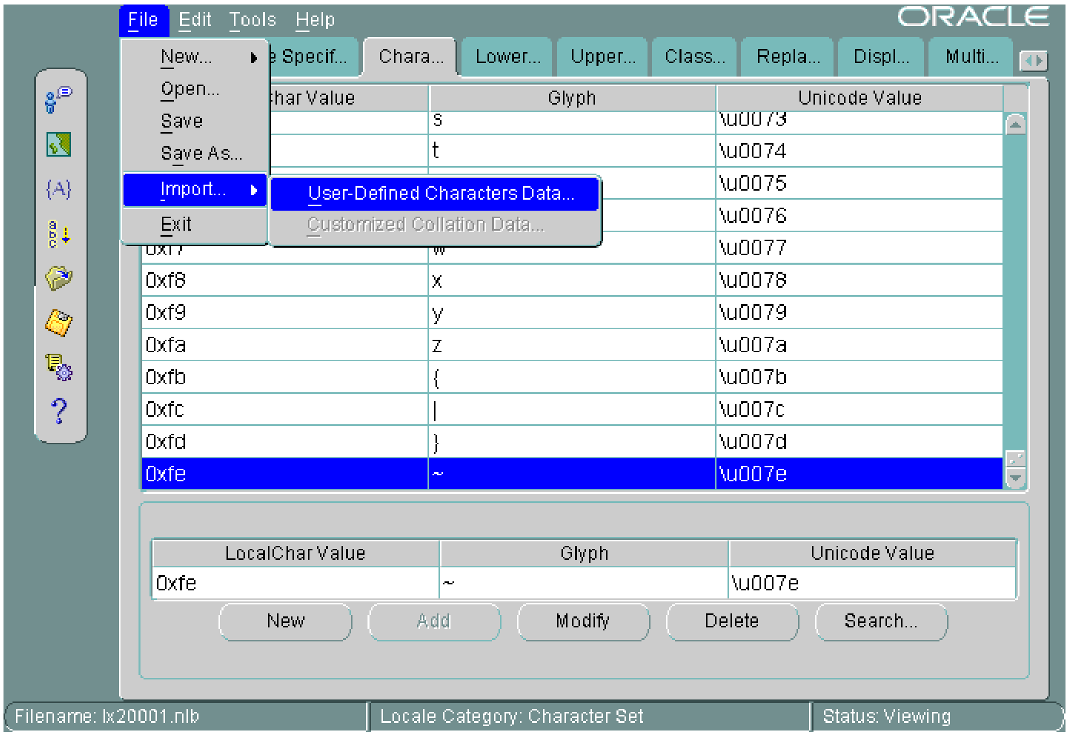
- `FIXED_WIDTH` is to identify character sets whose characters have a uniform length.
- `BYTE_UNIQUE` means the single-byte range of code points is distinct from the multibyte range. The code in the first byte indicates whether the character is single-byte or multibyte. An example is JA16EUC.
- `DISPLAY` identifies character sets that are used only for display on clients and not for storage. Some Arabic, Devanagari, and Hebrew character sets are display character sets.

- `SHIFT` is for character sets that require extra shift characters to distinguish between single-byte characters and multibyte characters.

  See Also: "Variable-width multibyte encoding schemes" on page 2-10 for more information about shift-in and shift-out character sets

Figure 12–23 shows how to add user-defined characters.

Figure 12–23 Importing User-Defined Character Data



Char Value	Glyph	Unicode Value
	s	\u0073
	t	\u0074
		\u0075
		\u0076
0xf7	w	\u0077
0xf8	x	\u0078
0xf9	y	\u0079
0xfa	z	\u007a
0xfb	{	\u007b
0xfc	|	\u007c
0xfd	}	\u007d
0xfe	~	\u007e

LocalChar Value	Glyph	Unicode Value
0xfe	~	\u007e

Filename: lx20001.nlb | Locale Category: Character Set | Status: Viewing

Open the Character Data Mapping screen. Highlight the character that you want to add characters after in the character set. In this example, the `0xfe` local character value is highlighted.

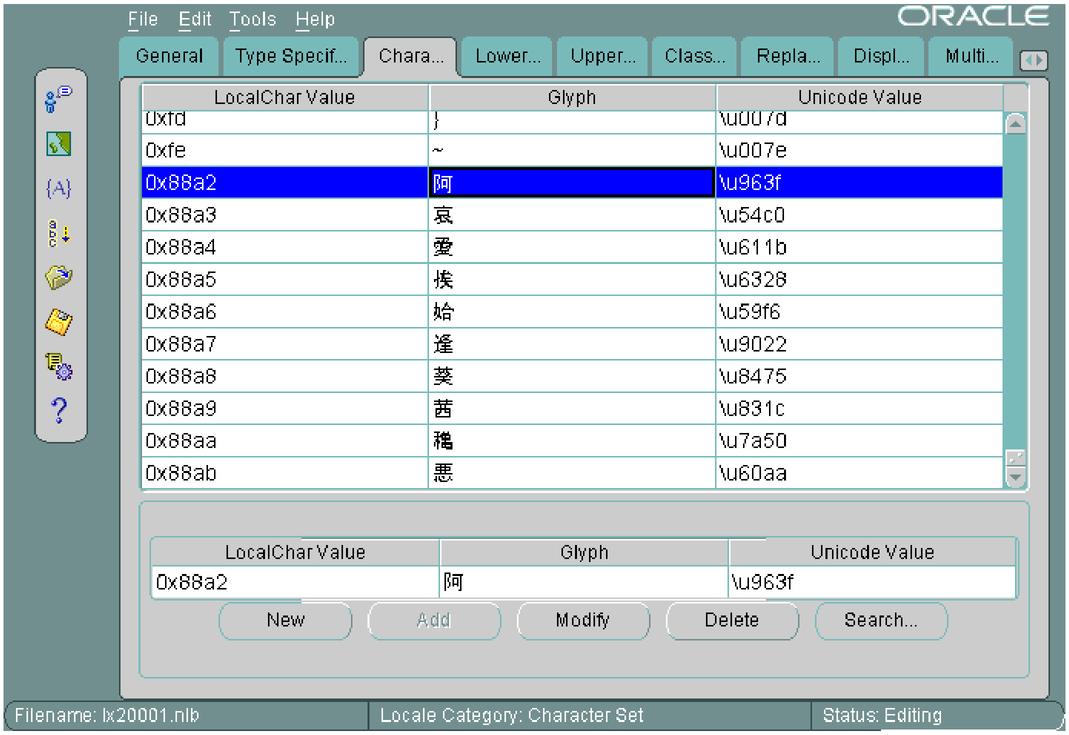
You can add one character at a time or use a text file to import a large number of characters. In this example, a text file is imported. The first column is the local character value. The second column is the Unicode value. The file contains the following character values:

88a2	963f
88a3	54c0
88a4	611b
88a5	6328
88a6	59f6
88a7	9022
88a8	8475
88a9	831c
88aa	7a50
88ab	60aa

In the File menu, choose Import User-Defined Customers Data.

Figure 12–24 shows that the imported characters are added after `0xfe` in the character set.

Figure 12–24   New Characters in the Character Set



LocalChar Value	Glyph	Unicode Value
0xfd	}	\u007d
0xfe	~	\u007e
0x88a2	阿	\u963f
0x88a3	哀	\u54c0
0x88a4	愛	\u611b
0x88a5	挨	\u6328
0x88a6	姶	\u59f6
0x88a7	逢	\u9022
0x88a8	葵	\u8475
0x88a9	茜	\u831c
0x88aa	穐	\u7a50
0x88ab	悪	\u60aa

LocalChar Value	Glyph	Unicode Value
0x88a2	阿	\u963f

Filename: lx20001.nlb | Locale Category: Character Set | Status: Editing

Supporting User-Defined Characters in Java

If you have Java products such as JDBC or SQLJ in your applications and want them to support user-defined characters, then customize your character set as desired. Then generate and install a special Java zip file (`gss_custom.zip`) into your Oracle home directory.

On UNIX, enter a command similar to the following:

```
$ORACLE_HOME/JRE/bin/jre -classpath $ORACLE_HOME/jlib/gss-1_1.zip:
    $ORACLE_HOME/jlib/gss_charset-1_2.zip Ginstall lx22710.nlt
```

On Windows, enter a command similar to the following:

```
%JREHOME%\bin\jre.exe -classpath %ORACLE_HOME%\jlib\gss-1_1.zip:
    %ORACLE_HOME%\jlib\gss_charset-1_2.zip  Ginstall lx22710.nlt
```

`%JREHOME%` is the `C:\Program Files\Oracle\jre\version_num` directory.

`lx22710.nlt` is an example of an NLT file created by customizing a character set using the Oracle Locale Builder.

These commands generate a `gss_custom.zip` file in the current directory. If you need to add support for more than one customized character set, you can append their definitions to the same `gss_custom.zip` file by re-issuing the command for each of the additional customized character sets. For example, enter the following commands on UNIX:

```
$ORACLE_HOME/JRE/bin/jre -classpath $ORACLE_HOME/jlib/gss-1_1.zip:
               $ORACLE_HOME/jlib/gss_charset-1_2.zip Ginstall lx22710.nlt


$ORACLE_HOME/JRE/bin/jre -classpath $ORACLE_HOME/jlib/gss-1_1.zip:
               $ORACLE_HOME/jlib/gss_charset-1_2.zip Ginstall lx22711.nlt

$ORACLE_HOME/JRE/bin/jre -classpath $ORACLE_HOME/jlib/gss-1_1.zip:
               $ORACLE_HOME/jlib/gss_charset-1_2.zip Ginstall lx22712.nlt
```

`lx22710.nlt`, `lx22711.nlt` and `lx22712.nlt` are contained in `gss_custom.zip`.

After `gss_custom.zip` has been created, store it in the `$ORACLE_HOME/ocommon/nls/admin/data` directory. Enter the following command:

```
% cp gss_custom.zip $ORACLE_HOME/ocommon/nls/admin/data
```

Adding the Custom Zip File to Java Components

You may want to add the `gss_custom.zip` file to the following Java components:

- Java Virtual Machine
- Oracle HTTP Server
- JDBC on the Client

Java Virtual Machine Load the zip file into the database.

Enter the following command on UNIX:

```
%loadjava -u sys/passwd -grant EXECUTE -synonym -r -r -v gss_custom.zip
```

Enter the following command on Windows:

```
loadjava -u sys/passwd -grant EXECUTE -synonym -r -r -v gss_custom.zip
```

Replace *`passwd`* by the password for `SYS`.

Oracle HTTP Server Edit the `jserv.properties` file.

On UNIX, add the following line:

```
wrapper.classpath = $ORACLE_HOME/ocommon/nls/admin/data/gss_custom.zip
```

On Windows, add the following line:

```
wrapper.classpath = %ORA_HOME%\ocommon\nls\admin\data\gss_custom.zip
```

JDBC on the Client Modify the `CLASSPATH`.

Enter the following command on UNIX:

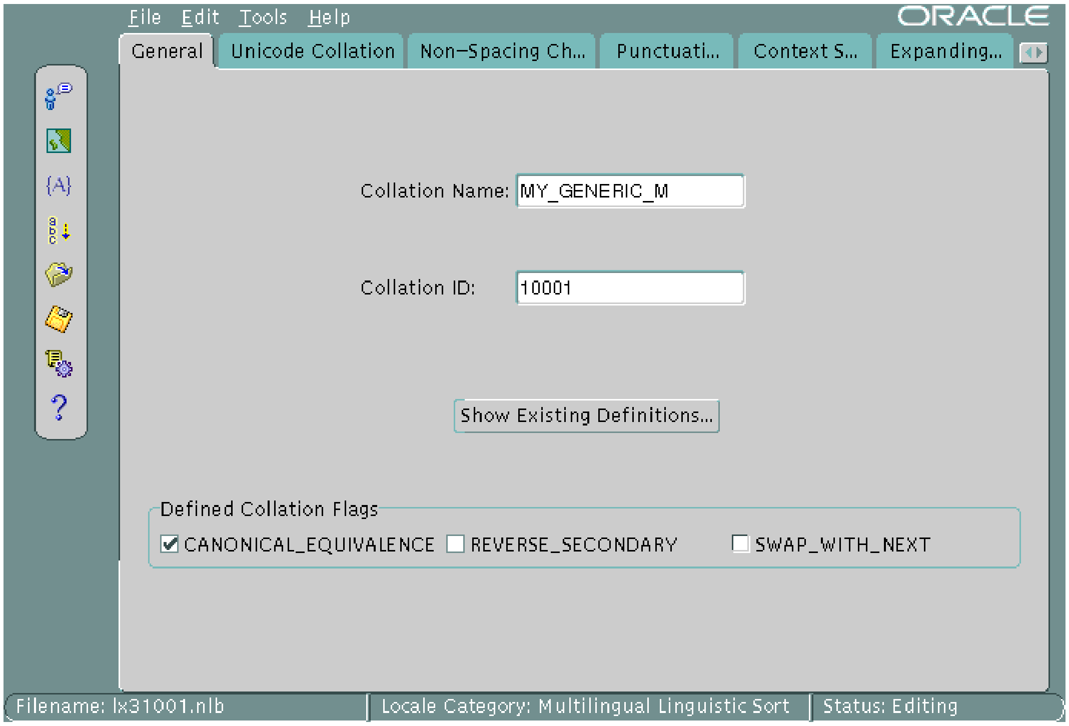
```
% setenv CLASSPATH $ORACLE_HOME/ocommon/nls/admin/data/gss_custom.zip
```

On Windows, add `%ORACLE_HOME%\ocommon\nls\admin\data\gss_custom.zip` to the existing `CLASSPATH`.

Creating a New Linguistic Sort with the Oracle Locale Builder

This section shows how to create a new multilingual linguistic sort called MY_GENERIC_M with a Collation ID of 10001. The GENERIC_M linguistic is used as the basis for the new linguistic sort. Figure 12–25 shows how to begin.

Figure 12–25 Collation General Information



Collation Name: MY_GENERIC_M

Collation ID: 10001

Defined Collation Flags

[✓ CANONICAL_EQUIVALENCE] REVERSE_SECONDARY SWAP_WITH_NEXT

Filename: lx31001.nlb | Locale Category: Multilingual Linguistic Sort | Status: Editing

Settings for the flags are automatically derived. SWAP_WITH_NEXT is relevant for Thai and Lao sorts. REVERSE_SECONDARY is for French sorts. CANONICAL_EQUIVALENCE determines whether canonical rules will be used. In this example, CANONICAL_EQUIVALENCE is checked.

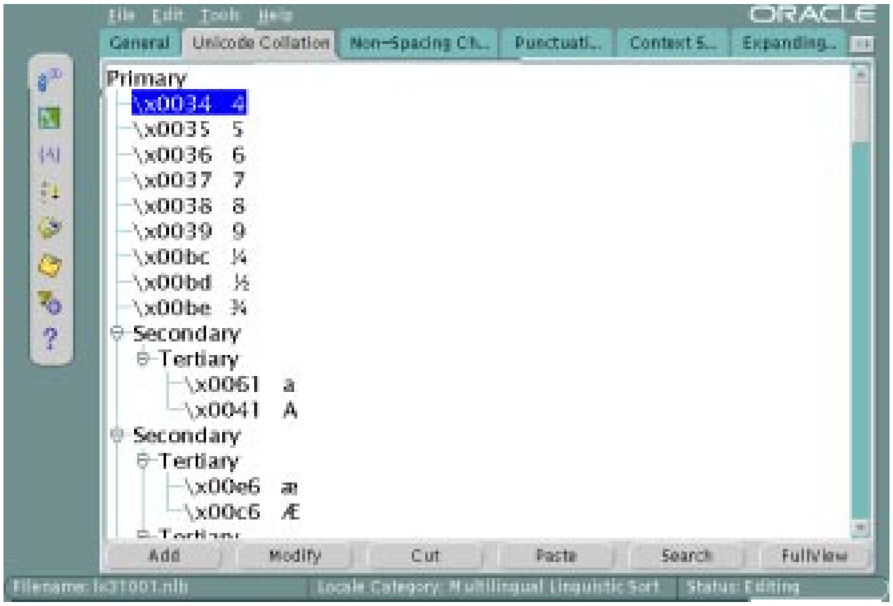
The valid range for Collation ID (sort ID) for a user-defined sort is 1,000 to 2,000 for monolingual collation and 10,000 to 11,000 for multilingual collation.

See Also:

- Figure 12–29, "Canonical Rules" for more information about canonical rules
- Chapter 4, "Linguistic Sorting"

Figure 12–26 shows the Unicode Collation Sequence screen.

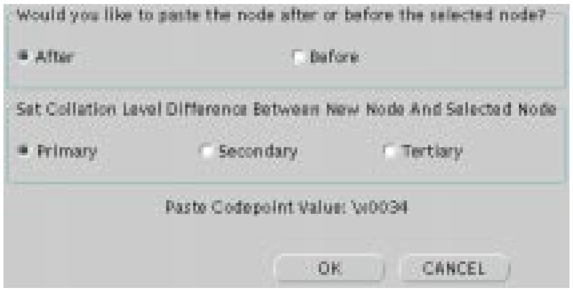
Figure 12–26 Unicode Collation Sequence



This example customizes the character set by moving digits so that they sort after letters. Complete the following steps:

1. Highlight the Unicode value that you want to move. In Figure 12–26, the `x0034` Unicode value is highlighted. Its location in the Unicode Collation Sequence is called a **node**.
2. Click Cut. Select the location where you want to move the node.
3. Click Paste. Clicking Paste opens the Paste Node dialog box, shown in Figure 12–27.

Figure 12–27 Paste Node Dialog Box



Would you like to paste the node after or before the selected node?

[✓ After] Before

Set Collation Level Difference Between New Node And Selected Node

[✓ Primary] Secondary Tertiary

Paste Codepoint Value: \u0034

4. The Paste Node dialog box enables you to choose whether to paste the node after or before the location you have selected. It also enables you to choose the level (Primary, Secondary, or Tertiary) of the node in relation to the node that you want to paste it next to.

   Select the position and the level at which you want to paste the node.

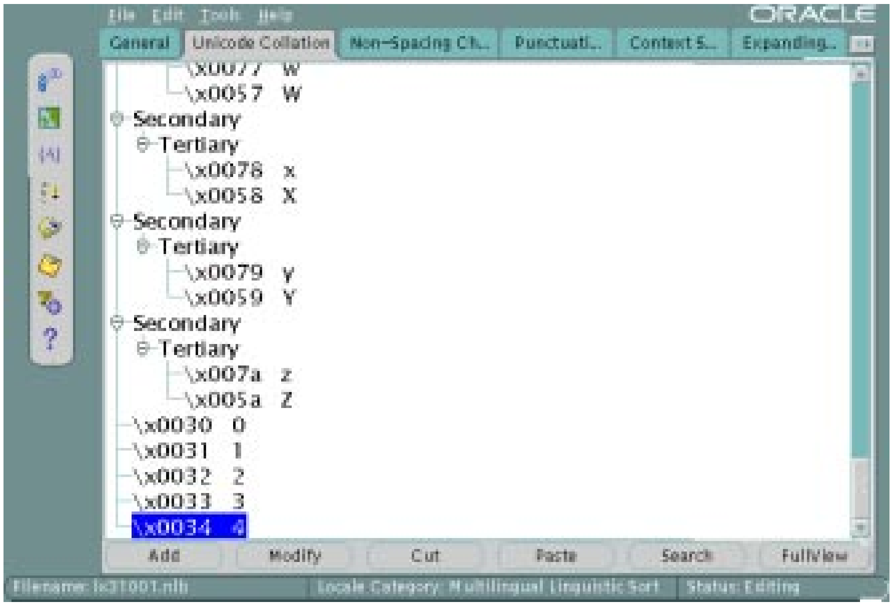
   In Figure 12–27, the After button and the Primary button are selected.

5. Click OK to paste the node.

Use similar steps to move other digits to a position after the letters `a` through `z`.

Figure 12–28 shows the resulting Unicode Collation Sequence after the digits `0` through `4` were moved to a position after the letters `a` through `z`.

Figure 12–28 Unicode Collation Sequence After Modification



```
\x0077 w
\x0057 W
Secondary
  Tertiary
    \x0078 x
    \x0058 X
Secondary
  Tertiary
    \x0079 y
    \x0059 Y
Secondary
  Tertiary
    \x007a z
    \x005a Z
\x0030 0
\x0031 1
\x0032 2
\x0033 3
\x0034 4
```

Filename: lx31001.nlb | Locale Category: Multilingual Linguistic Sort | Status: Editing

The rest of this section contains the following topics:

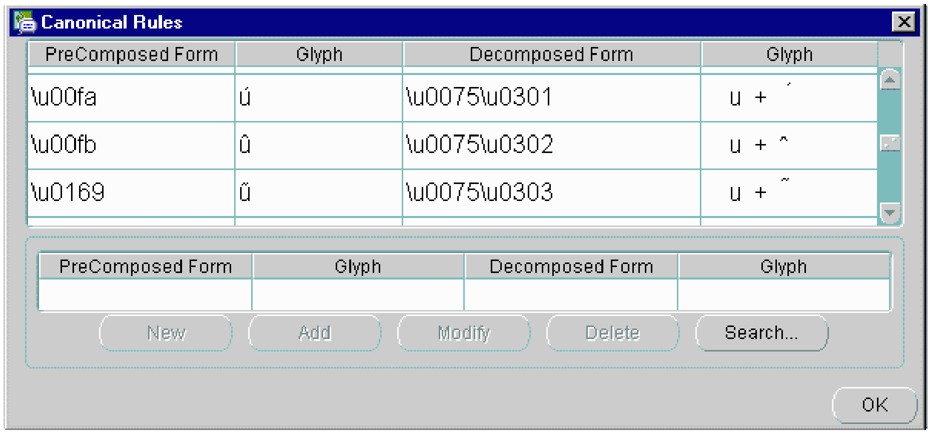
- Changing the Sort Order for All Characters with the Same Diacritic
- Changing the Sort Order for One Character with a Diacritic

Changing the Sort Order for All Characters with the Same Diacritic

This example shows how to change the sort order for characters with diacritics. You can do this by changing the sort for all characters containing a particular diacritic or by changing one character at a time. This example changes the sort of all characters with a circumflex (for example, û) to be after all characters containing a tilde.

Verify the current sort order by choosing Canonical Rules in the Tools menu. This opens the Canonical Rules dialog box, shown in Figure 12–29.

Figure 12–29 Canonical Rules



PreComposed Form	Glyph	Decomposed Form	Glyph
\u00fa	ú	\u0075\u0301	u + ´
\u00fb	û	\u0075\u0302	u + ^
\u0169	ũ	\u0075\u0303	u + ˜

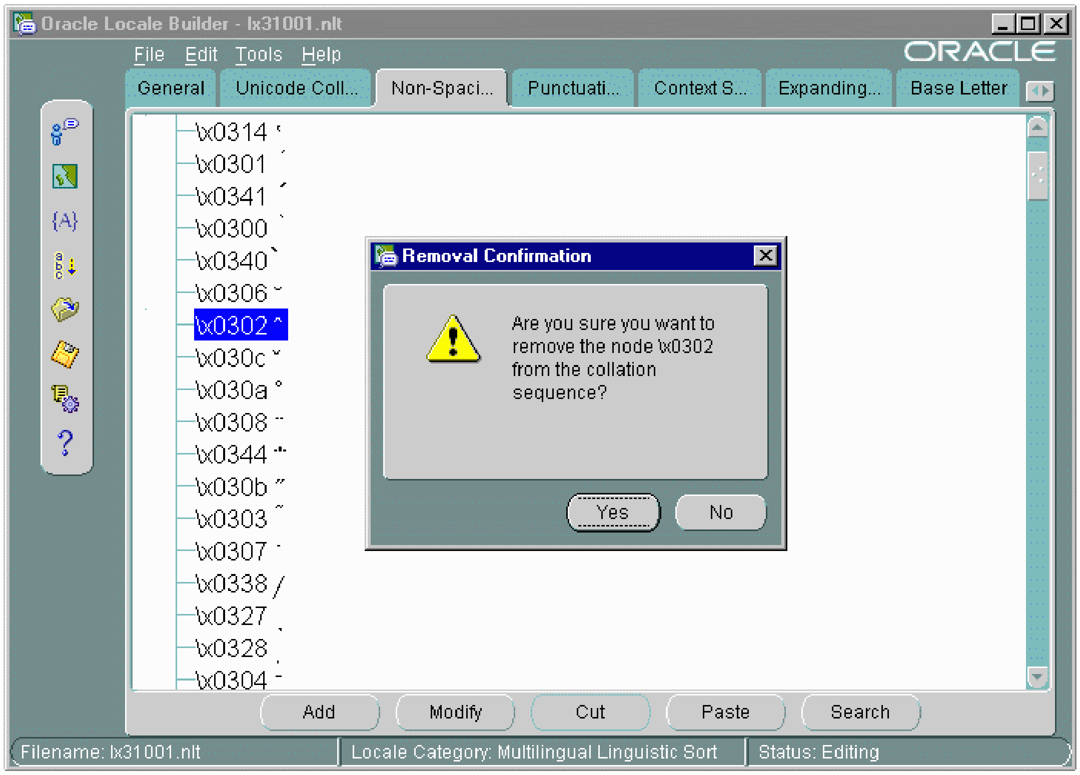
PreComposed Form	Glyph	Decomposed Form	Glyph

Figure 12–29 shows how characters are decomposed into their canonical equivalents and their current sorting orders. For example, û is represented as u plus ^.

> **See Also:** Chapter 4, "Linguistic Sorting" for more information about canonical rules

In the main Oracle Locale Builder window, click the Non-Spacing Characters tab. If you use the Non-Spacing Characters screen, then changes for diacritics apply to all characters. Figure 12–30 shows the Non-Spacing Characters screen.

Figure 12–30 Changing the Sort Order for All Characters with the Same Diacritic



Removal Confirmation

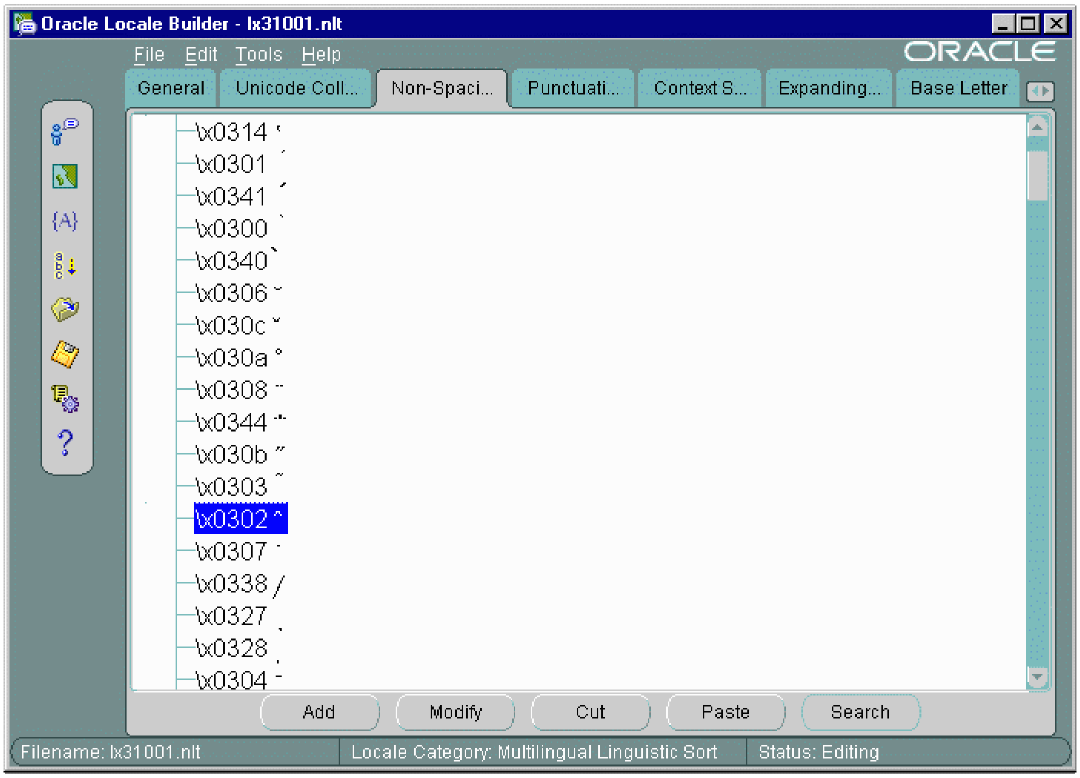
Are you sure you want to remove the node \x0302 from the collation sequence?

Filename: lx31001.nlt | Locale Category: Multilingual Linguistic Sort | Status: Editing

Select the circumflex and click Cut. Click Yes in the Removal Confirmation dialog box. Select the tilde and click Paste. Choose After and Secondary in the Paste Node dialog box and click OK.

Figure 12–31 illustrates the new sort order.

Figure 12–31 The New Sort Order for Characters with the Same Diacritic



\x0314
\x0301
\x0341
\x0300
\x0340
\x0306
\x030c
\x030a
\x0308
\x0344
\x030b
\x0303
\x0302
\x0307
\x0338
\x0327
\x0328
\x0304

Filename: lx31001.nlt
Locale Category: Multilingual Linguistic Sort
Status: Editing

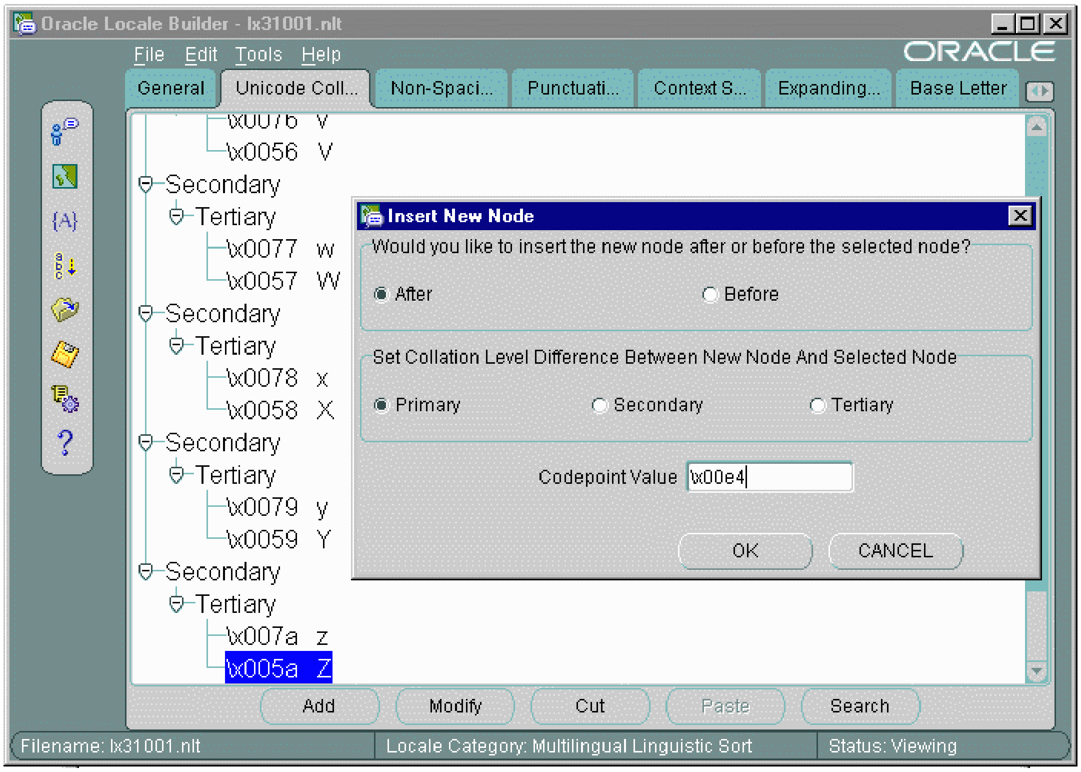
Changing the Sort Order for One Character with a Diacritic

To change the order of a specific character with a diacritic, insert the character directly into the appropriate position. Characters with diacritics do not appear in the Unicode Collation screen, so you cannot cut and paste them into the new location.

This example changes the sort order for ä so that it sorts after Z.

Select the Unicode Collation tab. Highlight the character, Z, that you want to put ä next to. Click Add. The Insert New Node dialog box appears, as shown in Figure 12–32.

Figure 12–32 Changing the Sort Order of One Character with a Diacritic



Insert New Node

Would you like to insert the new node after or before the selected node?

[✓ After] Before

Set Collation Level Difference Between New Node And Selected Node

[✓ Primary] Secondary Tertiary

Codepoint Value \x00e4

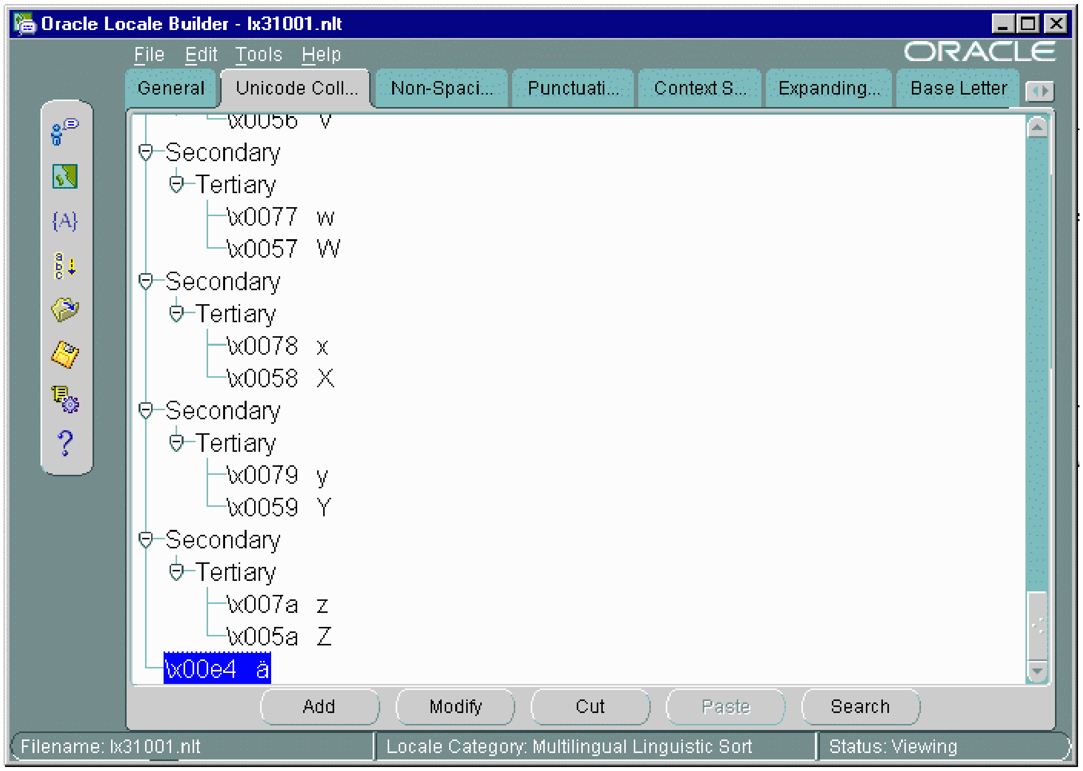
OK CANCEL

Filename: lx31001.nlt | Locale Category: Multilingual Linguistic Sort | Status: Viewing

Choose After and Primary in the Insert New Node dialog box. Enter the Unicode code point value of ä. The code point value is `\x00e4`. Click OK.

Figure 12–33 shows the resulting sort order.

Figure 12–33 New Sort Order After Changing a Single Character



```
\x0056  V
Secondary
  Tertiary
    \x0077  w
    \x0057  W
Secondary
  Tertiary
    \x0078  x
    \x0058  X
Secondary
  Tertiary
    \x0079  y
    \x0059  Y
Secondary
  Tertiary
    \x007a  z
    \x005a  Z
\x00e4  ä
```

Filename: lx31001.nlt | Locale Category: Multilingual Linguistic Sort | Status: Viewing

Generating and Installing NLB Files

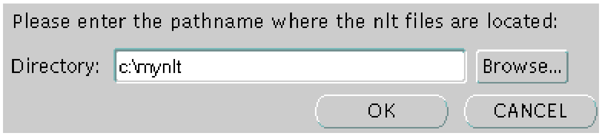
After you have defined a new language, territory, character set, or linguistic sort, generate new NLB files from the NLT files:

1. Back up the NLS installation boot file (`lx0boot.nlb`) and the NLS system boot file (`lx1boot.nlb`) in the ORA_NLS33 directory. On a UNIX platform, enter commands similar to the following:

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

2. In Oracle Locale Builder, choose Tools > Generate NLB or click the Generate NLB icon in the left side bar.
3. Click Browse to find the directory where the NLT file is located. The location dialog box is shown in Figure 12–34.

Figure 12–34 Location Dialog Box


Please enter the pathname where the nlt files are located:

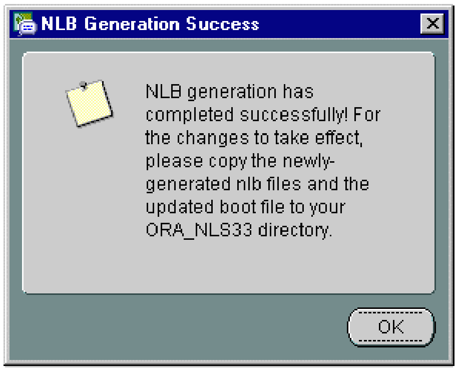
Directory: c:\mynlt

Do not try to specify an NLT file. Oracle Locale Builder generates an NLB file for each NLT file.

4. Click OK to generate the NLB files.

   Figure 12–35 illustrates the final notification that you have successfully generated NLB files for all NLT files in the directory.

Figure 12–35 NLB Generation Success Dialog Box


NLB Generation Success

NLB generation has completed successfully! For the changes to take effect, please copy the newly-generated nlb files and the updated boot file to your ORA_NLS33 directory.

5. Copy the `lx1boot.nlb` file into the path that is specified by the `ORA_NLS33` initialization parameter, typically `$ORACLE_HOME/OCOMMON/nls/admin/data`. For example, on a UNIX platform, enter a command similar to the following:

```
% cp /directory_name/lx1boot.nlb $ORA_NLS33/lx1boot.nlb
```

6. Copy the new NLB files into the ORA_NLS33 directory. For example, on a UNIX platform, enter commands similar to the following:

```
% cp /directory_name/lx22710.nlb $ORA_NLS33
% cp /directory_name/lx52710.nlb $ORA_NLA33
```

Note: Oracle Locale Builder generates NLB files in the directory where the NLT files reside.

7. Repeat the preceding steps on each hardware platform. NLB files are platform-specific binary files. You must compile and install the new NLB files on both the server and the client machines.
8. Restart the database to use the newly created locale data.
9. To use the new locale data on the client side, exit the client and re-invoke the client after installing the NLB files.

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 Sorting
- Calendar Systems
- Obsolete Locale Data

You can also obtain information about supported character sets, languages, territories, and sorting orders by querying the `V$NLS_VALID_VALUES` dynamic performance view.

> **See Also:** *Oracle9i Database Reference* for more information about the data that can be returned by this view

Languages

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

Table A–1 Oracle Supported Languages

Name	Abbreviation
AMERICAN	us
ARABIC	ar
ASSAMESE	as
BANGLA	bn
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
GUJARATI	gu
HEBREW	iw
HINDI	hi
HUNGARIAN	hu

Table A–1 Oracle Supported Languages (Cont.)

Name	Abbreviation
ICELANDIC	is
INDONESIAN	in
ITALIAN	i
JAPANESE	ja
KANNADA	kn
KOREAN	ko
LATIN AMERICAN SPANISH	esa
LATVIAN	lv
LITHUANIAN	lt
MALAY	ms
MALAYALAM	ml
MARATHI	mr
MEXICAN SPANISH	esm
NORWEGIAN	n
ORIYA	or
POLISH	pl
PORTUGUESE	pt
PUNJABI	pa
ROMANIAN	ro
RUSSIAN	ru
SIMPLIFIED CHINESE	zhs
SLOVAK	sk
SLOVENIAN	sl
SPANISH	e
SWEDISH	s
TAMIL	ta
TELUGU	te

Table A–1 Oracle Supported Languages (Cont.)

Name	Abbreviation
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
CANADIAN FRENCH	frc
CATALAN	ca
CZECH	cs
DANISH	dk
DUTCH	nl
FINNISH	sf
FRENCH	f
GERMAN	d
GREEK	el
HEBREW	iw
HUNGARIAN	hu
ITALIAN	i
JAPANESE	ja
KOREAN	ko

Table A–2 Oracle Supported Messages (Cont.)

Name	Abbreviation
LATIN AMERICAN SPANISH	esa
NORWEGIAN	n
POLISH	pl
PORTUGUESE	pt
ROMANIAN	ro
RUSSIAN	ru
SIMPLIFIED CHINESE	zhs
SLOVAK	sk
SPANISH	e
SWEDISH	s
THAI	th
TRADITIONAL CHINESE	zht
TURKISH	tr

Territories

Table A–3 lists the territories supported by the Oracle server.

Table A–3 Oracle Supported Territories

Name	Name	Name
ALGERIA	HONG KONG	PERU
AMERICA	HUNGARY	POLAND
AUSTRALIA	ICELAND	PORTUGAL
AUSTRIA	INDIA	PUERTO RICO
BAHRAIN	INDONESIA	QATAR
BANGLADESH	IRAQ	ROMANIA
BELGIUM	IRELAND	SAUDI ARABIA
BRAZIL	ISRAEL	SINGAPORE
BULGARIA	ITALY	SLOVAKIA

Table A–3 Oracle Supported Territories (Cont.)

Name	Name	Name
CANADA	JAPAN	SLOVENIA
CATALONIA	JORDAN	SOMALIA
CHILE	KAZAKHSTAN	SOUTH AFRICA
CHINA	KOREA	SPAIN
CIS	KUWAIT	SUDAN
COLOMBIA	LATVIA	SWEDEN
COSTA RICA	LEBANON	SWITZERLAND
CROATIA	LIBYA	SYRIA
CYPRUS	LITHUANIA	TAIWAN
CZECH REPUBLIC	LUXEMBOURG	THAILAND
DENMARK	MACEDONIA	THE NETHERLANDS
DJIBOUTI	MALAYSIA	TUNISIA
EGYPT	MAURITANIA	TURKEY
EL SALVADOR	MEXICO	UKRAINE
ESTONIA	MOROCCO	UNITED ARAB EMIRATES
FINLAND	NEW ZEALAND	UNITED KINGDOM
FRANCE	NICARAGUA	UZBEKISTAN
GUATEMALA	NORWAY	VENEZUELA
GERMANY	OMAN	VIETNAM
GREECE	PANAMA	YEMEN
-	-	YUGOSLAVIA

Character Sets

Oracle-supported character sets are listed in the following sections according to three broad language groups.

- Asian Language Character Sets
- European Language Character Sets

- Middle Eastern Language Character Sets

In addition, common subset/superset combinations are listed.

Note that some character sets may be listed under multiple language groups because they provide multilingual support. For instance, Unicode spans the Asian, European, and Middle Eastern language groups because it supports most of the major scripts of the world.

The comment section indicates the type of encoding used:

SB = Single-byte encoding

MB = Multibyte encoding

FIXED = Fixed-width multibyte encoding

As mentioned in Chapter 3, "Setting Up a Globalization Support Environment", the type of encoding affects performance, so use the most efficient encoding that meets your language needs. Also, some encoding types can only be used with certain data types. For instance, the AL16UTF16 character set can only be used as an `NCHAR` character set, and not as a database character set.

Also documented in the comment section are other unique features of the character set that may be important to users or your database administrator. For instance, whether the character set supports the new Euro currency symbol, whether user-defined characters are supported for character set customization, and whether the character set is a strict superset of ASCII (which will allow you to make use of the `ALTER DATABASE [NATIONAL] CHARACTER SET` statement in case of migration.)

EURO = Euro symbol supported

UDC = User-defined characters supported

ASCII = Strict superset of ASCII

Oracle does not document individual code page layouts. For specific details about a particular character set, its character repertoire, and code point values, you should refer to the actual national, international, or vendor-specific standards.

Asian Language Character Sets

Table A–4 lists the Oracle character sets that can support Asian languages.

Table A–4 Asian Language Character Sets

Name	Description	Comments
BN8BSCII	Bangladesh National Code 8-bit BSCII	SB, ASCII
ZHT16BIG5	BIG5 16-bit Traditional Chinese	MB, ASCII
ZHT16HKSCS	MS Windows Code Page 950 with Hong Kong Supplementary Character Set	MB, ASCII, EURO
ZHS16CGB231280	CGB2312-80 16-bit Simplified Chinese	MB, ASCII
ZHS32GB18030	GB18030-2000	MB, ASCII, EURO
JA16EUC	EUC 24-bit Japanese	MB, ASCII
JA16EUCTILDE	The same as JA16EUC except for the way that the wave dash and the tilde are mapped to and from Unicode.	MB, ASCII
JA16EUCYEN	EUC 24-bit Japanese with '\' mapped to the Japanese yen character	MB
ZHT32EUC	EUC 32-bit Traditional Chinese	MB, ASCII
ZHS16GBK	GBK 16-bit Simplified Chinese	MB, ASCII, UDC
ZHT16CCDC	HP CCDC 16-bit Traditional Chinese	MB, ASCII
JA16DBCS	IBM EBCDIC 16-bit Japanese	MB, UDC
JA16EBCDIC930	IBM DBCS Code Page 290 16-bit Japanese	MB, UDC
KO16DBCS	IBM EBCDIC 16-bit Korean	MB, UDC
ZHS16DBCS	IBM EBCDIC 16-bit Simplified Chinese	MB, UDC
ZHT16DBCS	IBM EBCDIC 16-bit Traditional Chinese	MB, UDC
KO16KSC5601	KSC5601 16-bit Korean	MB, ASCII
KO16KSCCS	KSCCS 16-bit Korean	MB, ASCII
JA16VMS	JVMS 16-bit Japanese	MB, ASCII
ZHS16MACCGB231280	Mac client CGB2312-80 16-bit Simplified Chinese	MB
JA16MACSJIS	Mac client Shift-JIS 16-bit Japanese	MB
TH8MACTHAI	Mac Client 8-bit Latin/Thai	SB
TH8MACTHAIS	Mac Server 8-bit Latin/Thai	SB, ASCII

Table A–4 Asian Language Character Sets (Cont.)

Name	Description	Comments
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
JA16SJISTILDE	The same as JA16SJIS except for the way that the wave dash and the tilde are mapped to and from Unicode.	MB, ASCII, UDC
JA16SJISYEN	Shift-JIS 16-bit Japanese with '\' mapped to the Japanese yen character	MB, UDC
ZHT32SOPS	SOPS 32-bit Traditional Chinese	MB, ASCII
ZHT16DBT	Taiwan Taxation 16-bit Traditional Chinese	MB, ASCII
TH8TISASCII	Thai Industrial Standard 620-2533 - ASCII 8-bit	SB, ASCII, EURO
TH8TISEBCDIC	Thai Industrial Standard 620-2533 - EBCDIC 8-bit	SB
ZHT32TRIS	TRIS 32-bit Traditional Chinese	MB, ASCII
AL16UTF16	See "Universal Character Sets" on page A-18 for details	MB, EURO, FIXED
AL32UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTFE	See "Universal Character Sets" on page A-18 for details	MB, EURO
VN8VN3	VN3 8-bit Vietnamese	SB, ASCII

European Language Character Sets

Table A–5 lists the Oracle character sets that can support European languages.

Table A–5 European Language Character Sets

Name	Description	Comments
US7ASCII	ASCII 7-bit American	SB, ASCII
SF7ASCII	ASCII 7-bit Finnish	SB
YUG7ASCII	ASCII 7-bit Yugoslavian	SB
RU8BESTA	BESTA 8-bit Latin / Cyrillic	SB, ASCII
EL8GCOS7	Bull EBCDIC GCOS7 8-bit Greek	SB
WE8GCOS7	Bull EBCDIC GCOS7 8-bit West European	SB
EL8DEC	DEC 8-bit Latin / Greek	SB
TR7DEC	DEC VT100 7-bit Turkish	SB
TR8DEC	DEC 8-bit Turkish	SB, ASCII
TR8EBCDIC1026	EBCDIC Code Page 1026 8-bit Turkish	SB
TR8EBCDIC1026S	EBCDIC Code Page 1026 Server 8-bit Turkish	SB
TR8PC857	IBM-PC Code Page 857 8-bit Turkish	SB, ASCII
TR8MACTURKISH	MAC Client 8-bit Turkish	SB
TR8MACTURKISHS	MAC Server 8-bit Turkish	SB, ASCII
TR8MSWIN1254	MS Windows Code Page 1254 8-bit Turkish	SB, ASCII, EURO
WE8BS2000L5	Siemens EBCDIC.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

Table A–5 European Language Character Sets (Cont.)

Name	Description	Comments
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
WE8EBCDIC284	EBCDIC Code Page 284 8-bit Latin American/Spanish	SB
WE8EBCDIC285	EBCDIC Code Page 285 8-bit West European	SB
WE8EBCDIC924	Latin 9 EBCDIC 924	SB, EBCDIC
WE8EBCDIC1047	EBCDIC Code Page 1047 8-bit West European	SB
WE8EBCDIC1047E	Latin 1/Open Systems 1047	SB, EBCDIC, EURO
WE8EBCDIC1140	EBCDIC Code Page 1140 8-bit West European	SB, EURO
WE8EBCDIC1140C	EBCDIC Code Page 1140 Client 8-bit West European	SB, EURO
WE8EBCDIC1145	EBCDIC Code Page 1145 8-bit West European	SB, EURO
WE8EBCDIC1146	EBCDIC Code Page 1146 8-bit West European	SB, EURO
WE8EBCDIC1148	EBCDIC Code Page 1148 8-bit West European	SB, EURO
WE8EBCDIC1148C	EBCDIC Code Page 1148 Client 8-bit West European	SB, EURO
F8EBCDIC297	EBCDIC Code Page 297 8-bit French	SB
WE8EBCDIC500C	EBCDIC Code Page 500 8-bit Oracle/c	SB
WE8EBCDIC500	EBCDIC Code Page 500 8-bit West European	SB
EE8EBCDIC870	EBCDIC Code Page 870 8-bit East European	SB
EE8EBCDIC870C	EBCDIC Code Page 870 Client 8-bit East European	SB
EE8EBCDIC870S	EBCDIC Code Page 870 Server 8-bit East European	SB
WE8EBCDIC871	EBCDIC Code Page 871 8-bit Icelandic	SB
EL8EBCDIC875	EBCDIC Code Page 875 8-bit Greek	SB
EL8EBCDIC875R	EBCDIC Code Page 875 Server 8-bit Greek	SB
CL8EBCDIC1025	EBCDIC Code Page 1025 8-bit Cyrillic	SB

Table A–5 European Language Character Sets (Cont.)

Name	Description	Comments
CL8EBCDIC1025C	EBCDIC Code Page 1025 Client 8-bit Cyrillic	SB
CL8EBCDIC1025R	EBCDIC Code Page 1025 Server 8-bit Cyrillic	SB
CL8EBCDIC1025S	EBCDIC Code Page 1025 Server 8-bit Cyrillic	SB
CL8EBCDIC1025X	EBCDIC Code Page 1025 (Modified) 8-bit Cyrillic	SB
BLT8EBCDIC1112	EBCDIC Code Page 1112 8-bit Baltic Multilingual	SB
BLT8EBCDIC1112S	EBCDIC Code Page 1112 8-bit Server Baltic Multilingual	SB
D8EBCDIC1141	EBCDIC Code Page 1141 8-bit Austrian German	SB, EURO
DK8EBCDIC1142	EBCDIC Code Page 1142 8-bit Danish	SB, EURO
S8EBCDIC1143	EBCDIC Code Page 1143 8-bit Swedish	SB, EURO
I8EBCDIC1144	EBCDIC Code Page 1144 8-bit Italian	SB, EURO
F8EBCDIC1147	EBCDIC Code Page 1147 8-bit French	SB, EURO
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

Table A–5 European Language Character Sets (Cont.)

Name	Description	Comments
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
EL8PC869	IBM-PC Code Page 869 8-bit Greek / Latin	SB, ASCII
LV8PC1117	IBM-PC Code Page 1117 8-bit Latvian	SB, ASCII
US8ICL	ICL EBCDIC 8-bit American	SB
WE8ICL	ICL EBCDIC 8-bit West European	SB
WE8ISOICLUK	ICL special version ISO8859-1	SB
WE8ISO8859P1	ISO 8859-1 West European	SB, ASCII
EE8ISO8859P2	ISO 8859-2 East European	SB, ASCII
SE8ISO8859P3	ISO 8859-3 South European	SB, ASCII
NEE8ISO8859P4	ISO 8859-4 North and North-East European	SB, ASCII
CL8ISO8859P5	ISO 8859-5 Latin / Cyrillic	SB, ASCII
AR8ISO8859P6	ISO 8859-6 Latin / Arabic	SB, ASCII
EL8ISO8859P7	ISO 8859-7 Latin / Greek	SB, ASCII, EURO
IW8ISO8859P8	ISO 8859-8 Latin / Hebrew	SB, ASCII
NE8ISO8859P10	ISO 8859-10 North European	SB, ASCII
BLT8ISO8859P13	ISO 8859-13 Baltic	SB, ASCII
CEL8ISO8859P14	ISO 8859-13 Celtic	SB, ASCII
WE8ISO8859P15	ISO 8859-15 West European	SB, ASCII, EURO
LA8ISO6937	ISO 6937 8-bit Coded Character Set for Text Communication	SB, ASCII

Table A–5 European Language Character Sets (Cont.)

Name	Description	Comments
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
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

Table A–5 European Language Character Sets (Cont.)

Name	Description	Comments
CL8ISOIR111	ISOIR111 Cyrillic	SB
CL8KOI8R	RELCOM Internet Standard 8-bit Latin / Cyrillic	SB, ASCII
CL8KOI8U	KOI8 Ukrainian Cyrillic	SB
US8BS2000	Siemens 9750-62 EBCDIC 8-bit American	SB
DK8BS2000	Siemens 9750-62 EBCDIC 8-bit Danish	SB
F8BS2000	Siemens 9750-62 EBCDIC 8-bit French	SB
D8BS2000	Siemens 9750-62 EBCDIC 8-bit German	SB
E8BS2000	Siemens 9750-62 EBCDIC 8-bit Spanish	SB
S8BS2000	Siemens 9750-62 EBCDIC 8-bit Swedish	SB
DK7SIEMENS9780X	Siemens 97801 / 97808 7-bit Danish	SB
F7SIEMENS9780X	Siemens 97801 / 97808 7-bit French	SB
D7SIEMENS9780X	Siemens 97801 / 97808 7-bit German	SB
I7SIEMENS9780X	Siemens 97801 / 97808 7-bit Italian	SB
N7SIEMENS9780X	Siemens 97801 / 97808 7-bit Norwegian	SB
E7SIEMENS9780X	Siemens 97801 / 97808 7-bit Spanish	SB
S7SIEMENS9780X	Siemens 97801 / 97808 7-bit Swedish	SB
EE8BS2000	Siemens EBCDIC.DF.04 8-bit East European	SB
WE8BS2000	Siemens EBCDIC.DF.04 8-bit West European	SB
WE8BS2000E	Siemens EBCDIC.DF.04 8-bit West European	SB, EURO
CL8BS2000	Siemens EBCDIC.EHC.LC 8-bit Cyrillic	SB
AL16UTF16	See "Universal Character Sets" on page A-18 for details	MB, EURO, FIXED
AL32UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTFE	See "Universal Character Sets" on page A-18 for details	MB, EURO

Middle Eastern Language Character Sets

Table A–6 lists the Oracle character sets that can support Middle Eastern languages.

Table A–6 Middle Eastern Character Sets

Name	Description	Comments
AR8APTEC715	APTEC 715 Server 8-bit Latin / Arabic	SB, ASCII
AR8APTEC715T	APTEC 715 8-bit Latin / Arabic	SB
AR8ASMO708PLUS	ASMO 708 Plus 8-bit Latin / Arabic	SB, ASCII
AR8ASMO8X	ASMO Extended 708 8-bit Latin / Arabic	SB, ASCII
AR8ADOS710	Arabic MS-DOS 710 Server 8-bit Latin / Arabic	SB, ASCII
AR8ADOS710T	Arabic MS-DOS 710 8-bit Latin / Arabic	SB
AR8ADOS720	Arabic MS-DOS 720 Server 8-bit Latin / Arabic	SB, ASCII
AR8ADOS720T	Arabic MS-DOS 720 8-bit Latin / Arabic	SB
TR7DEC	DEC VT100 7-bit Turkish	SB
TR8DEC	DEC 8-bit Turkish	SB
WE8EBCDIC37C	EBCDIC Code Page 37 8-bit Oracle / c	SB
IW8EBCDIC424	EBCDIC Code Page 424 8-bit Latin / Hebrew	SB
IW8EBCDIC424S	EBCDIC Code Page 424 Server 8-bit Latin / Hebrew	SB
WE8EBCDIC500C	EBCDIC Code Page 500 8-bit Oracle / c	SB
IW8EBCDIC1086	EBCDIC Code Page 1086 8-bit Hebrew	SB
AR8EBCDIC420S	EBCDIC Code Page 420 Server 8-bit Latin / Arabic	SB
AR8EBCDICX	EBCDIC 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

Table A–6 Middle Eastern Character Sets (Cont.)

Name	Description	Comments
IW8MACHEBREW	Mac Client 8-bit Hebrew	SB
AR8ARABICMAC	Mac Client 8-bit Latin / Arabic	SB
AR8ARABICMACT	Mac 8-bit Latin / Arabic	SB
TR8MACTURKISH	Mac Client 8-bit Turkish	SB
IW8MACHEBREWS	Mac Server 8-bit Hebrew	SB, ASCII
AR8ARABICMACS	Mac Server 8-bit Latin / Arabic	SB, ASCII
TR8MACTURKISHS	Mac Server 8-bit Turkish	SB, ASCII
TR8MSWIN1254	MS Windows Code Page 1254 8-bit Turkish	SB, ASCII, EURO
IW8MSWIN1255	MS Windows Code Page 1255 8-bit Latin / Hebrew	SB, ASCII, EURO
AR8MSWIN1256	MS Windows Code Page 1256 8-Bit Latin / Arabic	SB. ASCII, EURO
IN8ISCII	Multiple-Script Indian Standard 8-bit Latin / Indian Languages	SB
AR8MUSSAD768	Mussa'd Alarabi / 2 768 Server 8-bit Latin / Arabic	SB, ASCII
AR8MUSSAD768T	Mussa'd Alarabi / 2 768 8-bit Latin / Arabic	SB
AR8NAFITHA711	Nafitha Enhanced 711 Server 8-bit Latin / Arabic	SB, ASCII
AR8NAFITHA711T	Nafitha Enhanced 711 8-bit Latin / Arabic	SB
AR8NAFITHA721	Nafitha International 721 Server 8-bit Latin / Arabic	SB, ASCII
AR8NAFITHA721T	Nafitha International 721 8-bit Latin / Arabic	SB
AR8SAKHR706	SAKHR 706 Server 8-bit Latin / Arabic	SB, ASCII
AR8SAKHR707	SAKHR 707 Server 8-bit Latin / Arabic	SB, ASCII
AR8SAKHR707T	SAKHR 707 8-bit Latin / Arabic	SB
AR8XBASIC	XBASIC 8-bit Latin / Arabic	SB
WE8BS2000L5	Siemens EBCDIC.DF.04.L5 8-bit West European / Turkish	SB
AL16UTF16	See "Universal Character Sets" on page A-18 for details	MB, EURO, FIXED
AL32UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTF8	See "Universal Character Sets" on page A-18 for details	MB, ASCII, EURO
UTFE	See "Universal Character Sets" on page A-18 for details	MB, EURO

Universal Character Sets

Table A–7 lists the Oracle character sets that provide universal language support. They attempt to support all languages of the world, including, but not limited to, Asian, European, and Middle Eastern languages.

Table A–7 Universal Character Sets

Name	Description	Comments
AL16UTF16	Unicode 3.1 UTF-16 Universal character set	MB, EURO, FIXED
AL32UTF8	Unicode 3.1 UTF-8 Universal character set	MB, ASCII, EURO
UTF8	Unicode 3.0 UTF-8 Universal character set, CESU-8 compliant	MB, ASCII, EURO
UTFE	EBCDIC form of Unicode 3.0 UTF-8 Universal character set	MB, EURO

Note: CESU-8 defines an encoding scheme for Unicode that is identical to UTF-8 except for its representation of supplementary characters. In CESU-8, supplementary characters are represented as six-byte sequences that result from the transformation of each UTF-16 surrogate code unit into an eight-bit form that is similar to the UTF-8 transformation, but without first converting the input surrogate pairs to a scalar value. See Unicode Technical Report #26.

See Also: Chapter 5, "Supporting Multilingual Databases with Unicode"

Character Set Conversion Support

The following character set encodings are supported for conversion only. They cannot be used as the database or national character set:

- AL16UTF16LE
- ISO2022-CN
- ISO2022-JP
- ISO2022-KR
- HZ-GB-2312

You can use these character sets as the `source_char_set` or `dest_char_set` in the `CONVERT` function.

See Also:

- *Oracle9i SQL Reference* for more information about the `CONVERT` function
- "The CONVERT Function" on page 7-6

Subsets and Supersets

Table A–8 lists common subset / superset relationships.

Table A–8 Subset-Superset Pairs

Subset	Superset
AR8ADOS710	AR8ADOS710T
AR8ADOS720	AR8ADOS720T
AR8ADOS720T	AR8ADOS720
AR8APTEC715	AR8APTEC715T
AR8ARABICMACT	AR8ARABICMAC
AR8ISO8859P6	AR8ASMO708PLUS
AR8ISO8859P6	AR8ASMO8X
AR8MUSSAD768	AR8MUSSAD768T
AR8MUSSAD768T	AR8MUSSAD768
AR8NAFITHA711	AR8NAFITHA711T
AR8NAFITHA721	AR8NAFITHA721T
AR8SAKHR707	AR8SAKHR707T
AR8SAKHR707T	AR8SAKHR707
BLT8CP921	BLT8ISO8859P13
BLT8CP921	LT8MSWIN921
D7DEC	D7SIEMENS9780X
D7SIEMENS9780X	D7DEC
DK7SIEMENS9780X	N7SIEMENS9780X
I7DEC	I7SIEMENS9780X
I7SIEMENS9780X	IW8EBCDIC424

Table A–8 Subset-Superset Pairs (Cont.)

Subset	Superset
IW8EBCDIC424	IW8EBCDIC1086
KO16KSC5601	KO16MSWIN949
LT8MSWIN921	BLT8ISO8859P13
LT8MSWIN921	BLT8CP921
N7SIEMENS9780X	DK7SIEMENS9780X
US7ASCII	See Table A–9, "US7ASCII Supersets".
WE16DECTST	WE16DECTST2
WE16DECTST2	WE16DECTST
WE8DEC	TR8DEC
WE8DEC	WE8NCR4970
WE8ISO8859P1	WE8MSWIN1252
WE8ISO8859P9	TR8MSWIN1254
WE8NCR4970	TR8DEC
WE8NCR4970	WE8DEC
WE8PC850	WE8PC858
ZHS16GBK	ZHS32GB18030

US7ASCII is a special case because so many other character sets are supersets of it. Table A–9 lists supersets for US7ASCII.

Table A–9 US7ASCII Supersets

Supersets	Supersets	Supersets
AL24UTFFSS	EE8MACCES	NEE8ISO8859P4
AL32UTF8	EE8MACCROATIANS	RU8BESTA
AR8ADOS710	EE8MSWIN1250	RU8PC855
AR8ADOS710T	EE8PC852	RU8PC866
AR8ADOS720	EL8DEC	SE8ISO8859P3
AR8ADOS720T	EL8ISO8859P7	TH8MACTHAIS

Table A–9 US7ASCII Supersets (Cont.)

Supersets	Supersets	Supersets
AR8APTEC715	EL8MACGREEKS	TH8TISASCII
AR8APTEC715T	EL8MSWIN1253	TR8DEC
AR8ARABICMACS	EL8PC437S	TR8MACTURKISHS
AR8ASMO708PLUS	EL8PC851	TR8MSWIN1254
AR8ASMO8X	EL8PC869	TR8PC857
AR8HPARABIC8T	ET8MSWIN923	US8PC437
AR8ISO8859P6	HU8ABMOD	UTF8
AR8MSAWIN	HU8CWI2	VN8MSWIN1258
AR8MUSSAD768	IN8ISCII	VN8VN3
AR8MUSSAD768T	IS8PC861	WE8DEC
AR8NAFITHA711	IW8ISO8859P8	WE8DG
AR8NAFITHA711T	IW8MACHEBREWS	WE8ISO8859P1
AR8NAFITHA721	IW8MSWIN1255	WE8ISO8859P15
AR8NAFITHA721T	IW8PC1507	WE8ISO8859P9
AR8SAKHR706	JA16EUC	WE8MACROMAN8S
AR8SAKHR707	JA16SJIS	WE8MSWIN1252
AR8SAKHR707T	JA16TSTSET	WE8NCR4970
BG8MSWIN	JA16TSTSET2	WE8NEXTSTEP
BG8PC437S	JA16VMS	WE8PC850
BLT8CP921	KO16KSC5601	WE8PC858
BLT8ISO8859P13	KO16KSCCS	WE8PC860
BLT8MSWIN1257	KO16MSWIN949	WE8ROMAN8
BLT8PC775	KO16TSTSET	ZHS16CGB231280
BN8BSCII	LA8ISO6937	ZHS16GBK
CDN8PC863	LA8PASSPORT	ZHT16BIG5
CEL8ISO8859P14	LT8MSWIN921	ZHT16CCDC
CL8ISO8859P5	LT8PC772	ZHT16DBT

Table A–9 US7ASCII Supersets (Cont.)

Supersets	Supersets	Supersets
CL8KOI8R	LT8PC774	ZHT16HKSCS
CL8KOI8U	LV8PC1117	ZHT16MSWIN950
CL8ISOIR111	LV8PC8LR	ZHT32EUC
CL8MACCYRILLICS	LV8RST104090	ZHT32SOPS
CL8MSWIN1251	N8PC865	ZHT32TRIS
EE8ISO8859P2	NE8ISO8859P10	ZHS32GB18030
ZHT32EUCTST	-	-

Linguistic Sorting

Oracle offers two kinds of linguistic sorts, monolingual and multilingual. In addition, monolingual sorts can be extended to handle special cases. These special cases (represented with a prefix X) typically mean that the characters will be sorted differently from their ASCII values. For example, *ch* and *ll* are treated as a single character in XSPANISH.

Table A–10 lists the monolingual linguistic sorts supported by the Oracle server.

Table A–10 Monolingual Linguistic Sorts

Basic Name	Extended Name	Special Cases
ARABIC	-	-
ARABIC_MATCH	-	-
ARABIC_ABJ_SORT	-	-
ARABIC_ABJ_MATCH	-	-
ASCII7	-	-
BENGALI	-	-
BIG5	-	-
BINARY	-	-
BULGARIAN	-	-
CANADIAN FRENCH	-	-
CATALAN	XCATALAN	æ, AE, ß

Table A–10 Monolingual Linguistic Sorts (Cont.)

Basic Name	Extended Name	Special Cases
CROATIAN	XCROATIAN	D, L, N, d, l, n, ß
CZECH	XCZECH	ch, CH, Ch, ß
CZECH_PUNCTUTION	XCZECH_ PUNCTUATION	ch, CH, Ch, ß
DANISH	XDANISH	A, ß, Å, å
DUTCH	XDUTCH	ij, IJ
EBCDIC	-	-
EEC_EURO	-	-
EEC_EUROPA3	-	-
ESTONIAN	-	-
FINNISH	-	-
FRENCH	XFRENCH	-
GERMAN	XGERMAN	ß
GERMAN_DIN	XGERMAN_DIN	ß, ä, ö, ü, Ä, Ö, Ü
GBK	-	-
GREEK	-	-
HEBREW	-	-
HKSCS	-	-
HUNGARIAN	XHUNGARIAN	cs, gy, ny, sz, ty, zs, ß, CS, Cs, GY, Gy, NY, Ny, SZ, Sz, TY, Ty, ZS, Zs
ICELANDIC	-	-
INDONESIAN	-	-
ITALIAN	-	-
JAPANESE	-	-
LATIN	-	-
LATVIAN	-	-
LITHUANIAN	-	-
MALAY	-	-

Table A–10 Monolingual Linguistic Sorts (Cont.)

Basic Name	Extended Name	Special Cases
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

Table A–11 lists the multilingual linguistic sorts available in Oracle. All of them include `GENERIC_M` (an ISO standard for sorting Latin-based characters) as a base. Multilingual linguistic sorts are used for a specific primary language together with Latin-based characters. For example, KOREAN_M will sort Korean and Latin-based characters, but it will not collate Chinese, Thai, or Japanese characters.

Table A–11 Multilingual Linguistic Sorts

Basic Name	Explanation
CANADIAN_M	Canadian French sort supports reverse secondary, special expanding characters
DANISH_M	Danish sort supports sorting lower case characters before upper case characters

Table A–11 Multilingual Linguistic Sorts (Cont.)

Basic Name	Explanation
FRENCH_M	French sort supports reverse sort for secondary
GENERIC_M	Generic sorting order which is based on ISO14651 and Unicode canonical equivalence rules but excluding compatible equivalence rules
JAPANESE_M	Japanese sort supports SJIS character set order and EUC characters which are not included in SJIS
KOREAN_M	Korean sort: Hangul characters are based on Unicode binary order. Hanja characters based on pronunciation order. All Hangul characters are before Hanja characters
SPANISH_M	Traditional Spanish sort supports special contracting characters
THAI_M	Thai sort supports swap characters for some vowels and consonants
SCHINESE_RADICAL_M	Simplified Chinese sort based on radical as primary order and number of strokes order as secondary order
SCHINESE_STROKE_M	Simplified Chinese sort uses number of strokes as primary order and radical as secondary order
SCHINESE_PINYIN_M	Simplified Chinese PinYin sorting order
TCHINESE_RADICAL_M	Traditional Chinese sort based on radical as primary order and number of strokes order as secondary order
TCHINESE_STROKE_M	Traditional Chinese sort uses number of strokes as primary order and radical as secondary order. It supports supplementary characters.

Calendar Systems

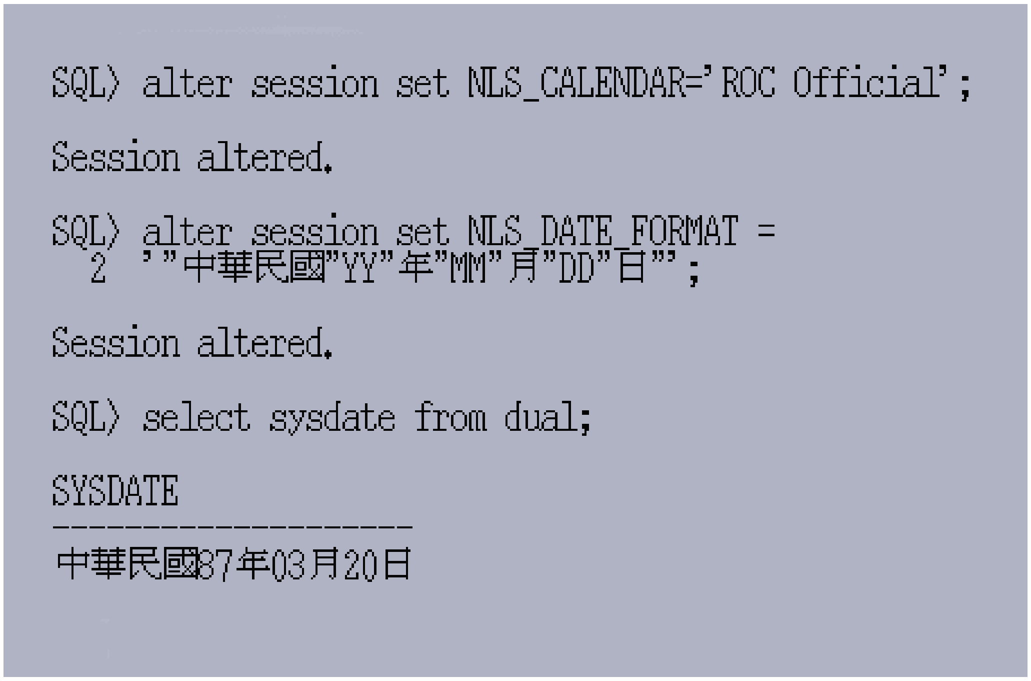
By default, most territory definitions use the Gregorian calendar system. Table A–12 lists the other calendar systems supported by the Oracle server.

Table A–12 Supported Calendar Systems

Name	Default Date Format	Character Set Used For Default Date Format
Japanese Imperial	EEYYMMDD	JA16EUC
ROC Official	EEyymmdd	ZHT32EUC
Thai Buddha	dd month EE yyyy	TH8TISASCII
Persian	DD Month YYYY	AR8ASMO8X
Arabic Hijrah	DD Month YYYY	AR8ISO8859P6
English Hijrah	DD Month YYYY	AR8ISO8859P6

Figure A–1 shows how March 20, 1998 appears in ROC Official:

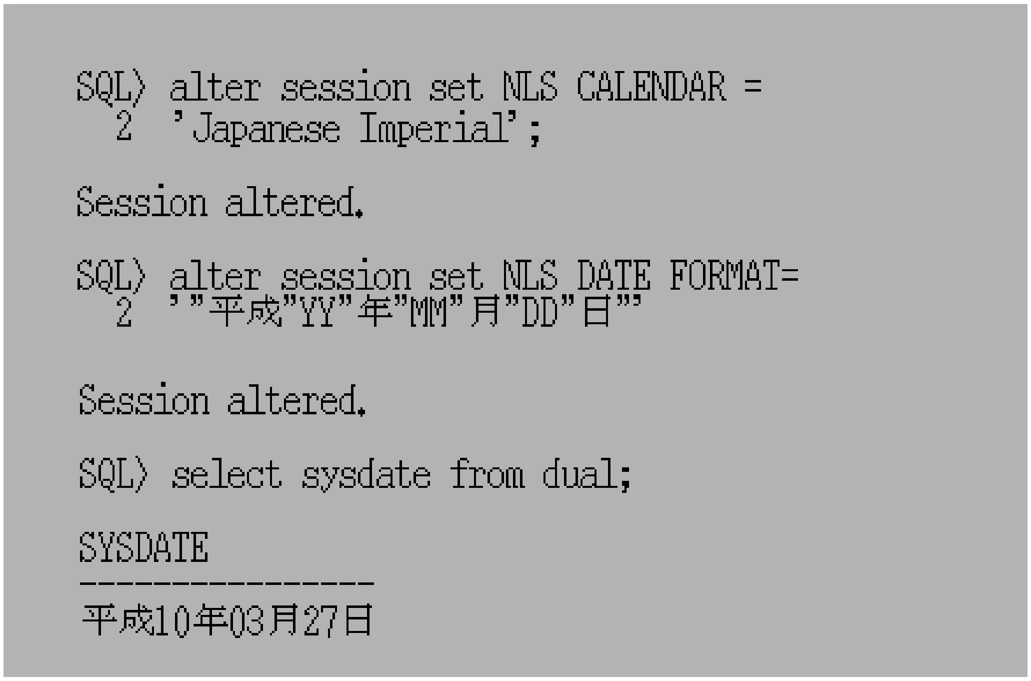
Figure A–1 ROC Official Example



```
SQL> alter session set NLS_CALENDAR='ROC Official';

Session altered.

SQL> alter session set NLS_DATE_FORMAT =
  2  '"中華民國"YY"年"MM"月"DD"日"';

Session altered.

SQL> select sysdate from dual;

SYSDATE
--------------------
中華民國87年03月20日
```

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

Figure A–2 Japanese Imperial Example



```
SQL> alter session set NLS_CALENDAR =
  2  'Japanese Imperial';

Session altered.

SQL> alter session set NLS_DATE_FORMAT=
  2  '"平成"YY"年"MM"月"DD"日"'

Session altered.

SQL> select sysdate from dual;

SYSDATE
----------------
平成10年03月27日
```

Obsolete Locale Data

Before Oracle server release 7.2, when a character set was renamed, the old name was usually supported along with the new name for several releases after the change. Beginning with release 7.2, the old names are no longer supported.

Table A–13 lists the affected character sets. If you reference any of these character sets in your code, replace them with their new name:

Table A–13 New Names for Obsolete Character Sets

Old Name	New Name
AL24UTFSS	UTF8, AL32UTF8

Table A–13 New Names for Obsolete Character Sets (Cont.)

Old Name	New Name
AR8MSAWIN	AR8MSWIN1256
CL8EBCDIC875S	CL8EBCDIC875R
EL8EBCDIC875S	EL8EBCDIC875R
JVMS	JA16VMS
JEUC	JA16EUC
SJIS	JA16SJIS
JDBCS	JA16DBCS
KSC5601	KO16KSC5601
KDBCS	KO16DBCS
CGB2312-80	ZHS16CGB231280
CNS 11643-86	ZHT32EUC
JA16EUCFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHS32EUCFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHS16GBKFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
JA16DBCSFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
KO16DBCSFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHS16DBCSFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHS16CGB231280 FIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHT16DBCSFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
KO16KSC5601FIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
JA16SJISFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.

Table A–13 New Names for Obsolete Character Sets (Cont.)

Old Name	New Name
ZHT16BIG5FIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.
ZHT32TRISFIXED	None. Replaced by new national character set. UTF8 and AL16UTF16.

Character set CL8MSWINDOW31 has been desupported. The newer character set CL8MSWIN1251 is actually a duplicate of CL8MSWINDOW31 and includes some characters omitted from the earlier version. Change any usage of CL8MSWINDOW31 to CL8MSWIN1251 instead.

AL24UTFFSS Character Set Desupported

The Unicode Character Set AL24UTFFSS has been desupported in Oracle9*i*. AL24UTFFSS was introduced with Oracle7 as the Unicode character set supporting UTF-8 encoding scheme based on the Unicode standard 1.1, which is now obsolete. In Oracle9*i*, Oracle now offers the Unicode database character set AL32UTF8 and UTF8, which includes the Unicode enhancements based on the Unicode standard 3.1.

The migration path for an existing AL24UTFFSS database is to upgrade to UTF8 prior to upgrading to Oracle9*i*. As with all migrations to a new database character set, Oracle Corporation recommends that you use the Character Set Scanner for data analysis before attempting to migrate your existing database character set to UTF8.

See Also: Chapter 11, "Character Set Scanner"

Bengali Language Definition Deprecated

The Bengali language definition is not compatible with Unicode standards. Oracle Corporation recommends that customers use the Bangla language definition instead. Bangla was introduced in Oracle9*i* Database Release 1 (9.0.1).

The Bengali language definition is supported in Oracle9*i* Database Release 2 (9.2), but it may be desupported in a future release.

Czechoslovakia Territory Definition Deprecated

Oracle Corporation recommends that customers use either Czech Republic or Slovakia territory definitions in Oracle9*i* Database Release 2 (9.2). The Czechoslovakia territory definition is supported in Oracle9*i* Database Release 2 (9.2), but it may be desupported in a future release.

B

Unicode Character Code Assignments

This appendix offers an introduction to how Unicode assigns characters. This appendix contains:

Unicode Code Ranges

Table B–1 contains code ranges that have been allocated in Unicode for UTF-16 character codes.

Table B–1 Unicode Character Code Ranges for UTF-16 Character Codes

Types of Characters	First 16 Bits	Second 16 Bits
ASCII	0000-007F	-
European (except ASCII), Arabic, Hebrew	0080-07FF	-
indic, Thai, certain symbols (such as the euro symbol), Chinese, Japanese, Korean	0800-0FFF	-
	1000 - CFFF	
	D000 - D7FF	
	F900 - FFFF	
Private Use Area #1	E000 - EFFF	-
	F000 - F8FF	
Supplementary characters: Additional Chinese, Japanese, and Korean characters; historic characters; musical symbols; mathematical symbols	D800 - D8BF	DC00 - DFFF
	D8CO - DABF	DC00 - DFFF
	DAC0 - DB7F	DC00 - DFFF
rivate Use Area #2	DB80 - DBBF	DC00 - DFFF
	DBC0 - DBFF	DC00 - DFFF

Table B–2 contains code ranges that have been allocated in Unicode for UTF-8 character codes.

Table B–2 Unicode Character Code Ranges for UTF-8 Character Codes

Types of Characters	First Byte	Second Byte	Third Byte	Fourth Byte
ASCII	00 - 7F	-	-	-
European (except ASCII), Arabic, Hebrew	C2 - DF	80 - BF	-	-

Table B–2 Unicode Character Code Ranges for UTF-8 Character Codes (Cont.)

Types of Characters	First Byte	Second Byte	Third Byte	Fourth Byte
Iindic, Thai, certain symbols (such as the euro symbol), Chinese, Japanese, Korean	E0	A0 - BF	80 - BF	-
	E1 - EC	80 - BF	80 - BF	
	ED	80 - 9F	80 - BF	
	EF	A4 - BF	80 - BF	
Private Use Area #1	EE	80 - BF	80 - BF	-
	EF	80 - A3	80 - BF	
Supplementary characters: Additional Chinese, Japanese, and Korean characters; historic characters; musical symbols; mathematical symbols	F0	90 - BF	80 - BF	80 - BF
	F1 - F2	80 - BF	80 - BF	80 - BF
	F3	80 - AF	80 - BF	80 - BF
Private Use Area #2	F3	B0 - BF	80 - BF	80 - BF
	F4	80 - 8F	80 - BF	80 - BF

Note: Blank spaces represent non-applicable code assignments. Character codes are shown in hexadecimal representation.

UTF-16 Encoding

As shown in Table B–1, UTF-16 character codes for some characters (Additional Chinese / Japanese / Korean characters and Private Use Area #2) are represented in two units of 16-bits. These are supplementary characters. A supplementary character consists of two 16-bit values. The first 16-bit value is encoded in the range from 0xD800 to 0xDBFF. The second 16-bit value is encoded in the range from 0xDC00 to 0xDFFF. With supplementary characters, UTF-16 character codes can represent more than one million characters. Without supplementary characters, only 65,536 characters can be represented. Oracle's AL16UTF16 character set supports supplementary characters.

See Also: "Supplementary Characters" on page 5-3

UTF-8 Encoding

The UTF-8 character codes in Table B–2 show that the following conditions are true:

- ASCII characters use 1 byte
- European (except ASCII), Arabic, and Hebrew characters require 2 bytes
- Indic, Thai, Chinese, Japanese, and Korean characters as well as certain symbols such as the euro symbol require 3 bytes
- Characters in the Private Use Area #1 require 3 bytes
- Supplementary characters require 4 bytes
- Characters in the Private Use Area #2 require 4 bytes

Oracle's AL32UTF8 character set supports 1-byte, 2-byte, 3-byte, and 4-byte values. Oracle's UTF8 character set supports 1-byte, 2-byte, and 3-byte values, but not 4-byte values.

Glossary

AL16UTF16

The default Oracle character set for the SQL NCHAR data type, which is used for the national character set. It encodes Unicode data in the UTF-16 encoding.

> **See Also:** national character set

AL32UTF8

An Oracle character set for the SQL CHAR data type, which is used for the database character set. It encodes Unicode data in the UTF-8 encoding.

> **See Also:** database character set

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 is US7ASCII.

binary sorting

Ordering character strings based on their binary coded values.

byte semantics

Treatment of strings as a sequence of bytes.

> **See Also:** character semantics and length semantics

canonical equivalence

A basic equivalence between characters or sequences of characters. For example, ç is equivalent to the combination of c and , . They cannot be distinguished when they are correctly rendered.

case

Refers to the condition of being uppercase or lowercase. For example, in a Latin alphabet, A is the uppercase glyph for a, the lowercase glyph.

case conversion

Changing a character from uppercase to lowercase or vice versa.

character

A character is an abstract element of text. A character is different from a glyph, which is a specific representation of a character. For example, the first character of the English upper-case alphabet can be displayed as A, A, A, and so on. These forms are different glyphs that represent the same character. A character, a character code, and a glyph are related as follows:

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

For example, the first character of the English uppercase alphabet is represented in computer memory as a number. The number is called the **encoding** or the **character code**. The character code for the first character of the English uppercase alphabet is 0x41 in the ASCII encoding scheme. The character code is 0xc1 in the EBCDIC encoding scheme.

You must choose a font to display or print the character. The available fonts depend on which encoding scheme is being used. The character can be printed or displayed as A, A, or A, for example. The forms are different **glyph**s that represent the same character.

See Also: character code and glyph

character code

A character code is a number that represents a specific character. The number depends on the encoding scheme. For example, the character code of the first character of the English uppercase alphabet is 0x41 in the ASCII encoding scheme, but it is 0xc1 in the EBCDIC encoding scheme.

See Also: character

character semantics

Treatment of strings as a sequence of characters.

See Also: byte semantics and length semantics

character set

A collection of elements that represent textual information for a specific language or group of languages. One language can be represented by more than one character set.

A character set does not always imply a specific character encoding scheme. A character encoding scheme is the assignment of a character code to each character in a character set.

In this manual, a character set usually does imply a specific character encoding scheme. Therefore, a character set is the same as an encoded character set in this manual.

character set migration

Changing the character set of an existing database.

character string

An ordered group of characters.

A character string can also contain no characters. In this case, the character string is called a **null string**. The number of characters in a null string is 0 (zero).

character classification

Character classification information provides details about the type of character associated with each character code. For example, a character can uppercase, lowercase, punctuation, or control character.

character encoding scheme

A rule that assigns numbers (character codes) to all characters in a character set. **Encoding scheme**, **encoding method**, and **encoding** also mean **character encoding scheme**.

client character set

The encoded character set used by the client. A client character set can differ from the server character set. The server character set is called the **database character set**.

If the client character set is different from the database character set, then character set conversion must occur.

See Also: database character set

code point

The numeric representation of a character in a character set. For example, the code point of `A` in the ASCII character set is 0x41. The code point of a character is also called the **encoded value** of a character.

See Also: Unicode code point

collation

Ordering of character strings according to rules about sorting characters that are associated with a language in a specific locale. Also called **linguistic sort**.

See Also:

- linguistic sort
- monolingual linguistic sort
- multilingual linguistic sort

data scanning

The process of identifying potential problems with character set conversion and truncation of data before migrating the database character set.

database character set

The encoded character set that is used to store text in the database. This includes `CHAR`, `VARCHAR2`, `LONG`, and fixed-width `CLOB` column values and all SQL and PL/SQL text.

diacritic

A mark near or through a character or combination of characters that indicates a different sound than the sound of the character without the diacritical mark. For example, the cedilla in `façade` is a diacritic. It changes the sound of `c`.

EBCDIC

Extended Binary Coded Decimal Interchange Code. EBCDIC is a family of encoded character sets used mostly on IBM systems.

encoded character set

A character set with an associated character encoding scheme. An encoded character set specifies the number (character code) that is assigned to each character.

See Also: character encoding scheme

encoded value

The numeric representation of a character in a character set. For example, the code point of A in the ASCII character set is 0x41. The encoded value of a character is also called the **code point** of a character.

font

An ordered collection of character glyphs that provides a graphical representation of characters in a character set.

globalization

The process of making software suitable for different linguistic and cultural environments. Globalization should not be confused with localization, which is the process of preparing software for use in one specific locale.

glyph

A glyph (font glyph) is a specific representation of a character. A character can have many different glyphs. For example, the first character of the English uppercase alphabet can be printed or displayed as A, A, A, and so on.

These forms are different glyphs that represent the same character.

See Also: character

ideograph

A symbol that represents an idea. Chinese is an example of an ideographic writing system.

ISO

International Organization for Standards. A worldwide federation of national standards bodies from 130 countries. The mission of ISO is to develop and promote standards in the world to facilitate the international exchange of goods and services.

ISO 8859

A family of 8-bit encoded character sets. The most common one is ISO 8859-1 (also known as ISO Latin1), and is used for Western European languages.

ISO 14651

A multilingual linguistic sort standard that is designed for almost all languages of the world.

See Also: multilingual linguistic sort

ISO/IEC 10646

A universal character set standard that defines the characters of most major scripts used in the modern world. In 1993, ISO adopted Unicode version 1.1 as ISO/IEC 10646-1:1993. ISO/IEC 10646 has two formats: UCS-2 is a 2-byte fixed-width format, and UCS-4 is a 4-byte fixed-width format. There are three levels of implementation, all relating to support for composite characters:

- Level 1 requires no composite character support.
- Level 2 requires support for specific scripts (including most of the Unicode scripts such as Arabic and Thai).
- Level 3 requires unrestricted support for composite characters in all languages.

ISO currency

The 3-letter abbreviation used to denote a local currency, based on the ISO 4217 standard. For example, USD represents the United States dollar.

ISO Latin1

The ISO 8859-1 character set standard. It is an 8-bit extension to ASCII that adds 128 characters that include the most common Latin characters used in Western Europe. The Oracle character set name is WE8ISO8859P1.

See Also: ISO 8859

length semantics

Length semantics determines how you treat the length of a character string. The length can be treated as a sequence of characters or bytes.

See Also: character semantics and byte semantics

linguistic index

An index built on a linguistic sort order.

linguistic sort

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

> **See Also:** multilingual linguistic sort and monolingual linguistic sort

locale

A collection of information about 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-specific or culture-specific information for software systems. Translation of an application's user interface is an example of localization. Localization should not be confused with globalization, which is the making software suitable for different linguistic and cultural environments.

monolingual linguistic sort

An Oracle sort that has two levels of comparison for strings. Most European languages can be sorted with a monolingual sort, but it is inadequate for Asian languages.

> **See Also:** multilingual linguistic sort

monolingual support

Support for only one language.

multibyte

Two or more bytes.

When character codes are assigned to all characters in a specific language or a group of languages, one byte (8 bits) can represent 256 different characters. Two bytes (16 bits) can represent up to 65,536 different characters. Two bytes are not enough to represent all the characters for many languages. Some characters require 3 or 4 bytes.

One example is the UTF8 Unicode encoding. In UTF8, there are many 2-byte and 3-byte characters.

Another example is Traditional Chinese, used in Taiwan. It has more than 80,000 characters. Some character encoding schemes that are used in Taiwan use 4 bytes to encode characters.

See Also: single byte

multibyte character

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 codes under different encoding schemes. Oracle cannot tell if a character is a multibyte character without knowing which character encoding scheme is being used. For example, Japanese Hankaku-Katakana (half-width Katakana) characters are one byte in the JA16SJIS encoded character set, two bytes in JA16EUC, and three bytes in UTF8.

See Also: single-byte character

multibyte character string

A character string that consists of one of the following:

- No characters (called a **null string**)
- One or more single-byte characters
- A mixture of one or more single-byte characters and one or more multibyte characters
- One or more multibyte characters

multilingual linguistic sort

An Oracle sort that uses evaluates strings on three levels. Asian languages require a multilingual linguistic sort even if data exists in only one language. Multilingual linguistic sorts are also used when data exists in several languages.

national character set

An alternate character set from the database character set that can be specified for `NCHAR`, `NVARCHAR2`, and `NCLOB` columns. National character sets are in Unicode only.

NLB files

Binary files used by the Locale Builder to define locale-specific data. They define all of the locale definitions that are shipped with a specific release of the Oracle database server. You can create user-defined NLB files with Oracle Locale Builder.

See Also: Oracle Locale Builder and NLT files

NLS

National Language Support. NLS allows users to interact with the database in their native languages. It also allows applications to run in different linguistic and cultural environments. The term is somewhat obsolete because Oracle supports global users at one time.

NLSRTL

National Language Support Runtime Library. This library is responsible for providing locale-independent algorithms for internationalization. The locale-specific information (that is, NLSDATA) is read by the NLSRTL library during run-time.

NLT files

Text files used by the Locale Builder to define locale-specific data. Because they are in text, you can view the contents.

null string

A character string that contains no characters.

Oracle Locale Builder

A GUI utility that offers a way to view, modify, or define locale-specific data. You can also create your own formats for language, territory, character set, and linguistic sort.

replacement character

A character used during character conversion when the source character is not available in the target character set. For example, ? is often used as Oracle's default replacement character.

restricted multilingual support

Multilingual support that is restricted to a group of related languages.Western European languages can be represented with ISO 8859-1, for example. If multilingual support is restricted, then Thai could not be added to the group.

SQL CHAR datatypes

Includes `CHAR`, `VARCHAR`, `VARCHAR2`, `CLOB`, and `LONG` datatypes.

SQL NCHAR datatypes

Includes `NCHAR`, `NVARCHAR`, `NVARCHAR2`, and `NCLOB` datatypes.

script

A collection of related graphic symbols that are used in a writing system. Some scripts can represent multiple languages, and some languages use multiple scripts. Example of scripts include Latin, Arabic, and Han.

single byte

One byte. One byte usually consists of 8 bits. When character codes are assigned to all characters for a specific language, one byte (8 bits) can represent 256 different characters.

See Also: multibyte

single-byte character

A single-byte character is a character whose character code consists of one byte under a specific character encoding scheme. Note that the same character may have different character codes under different encoding schemes. Oracle cannot tell which character is a single-byte character without knowing which encoding scheme is being used. For example, the euro currency symbol is one byte in the WE8MSWIN1252 encoded character set, two bytes in AL16UTF16, and three bytes in UTF8.

See Also: multibyte character

single-byte character string

A single-byte character string is a character string that consists of one of the following:

- No character (called a **null string**)
- One or more single-byte characters

supplementary characters

The first version of Unicode was a 16-bit, fixed-width encoding that used two bytes to encode each character. This allowed 65,536 characters to be represented. However, more characters need to be supported because of the large number of Asian ideograms.

Unicode 3.1 defines supplementary characters to meet this need. It uses two 16-bit code points (also known as **surrogate pairs**) to represent a single character. This allows an additional 1,048,576 characters to be defined. The Unicode 3.1 standard added the first group of 44,944 supplementary characters.

surrogate pairs

See Also: supplementary characters

syllabary

Provide a mechanism for communicating phonetic information along with the ideographic characters used by languages such as Japanese.

UCS-2

A 1993 ISO / IEC standard character set. It is a fixed-width, 16-bit Unicode character set. Each character occupies 16 bits of storage. The ISO Latin1 characters are the first 256 code points, so it can be viewed as a 16-bit extension of ISO Latin1.

UCS-4

A fixed-width, 32-bit Unicode character set. Each character occupies 32 bits of storage. The UCS-2 characters are the first 65,536 code points in this standard, so it can be viewed as a 32-bit extension of UCS-2. This is also sometimes referred to as ISO-10646.

Unicode

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

Unicode database

A database whose database character set is UTF-8.

Unicode code point

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.

Unicode datatype

A SQL `NCHAR` datatype (`NCHAR`, `NVARCHAR2`, and `NCLOB`). You can store Unicode characters in columns of these datatypes even if the database character set is not Unicode.

unrestricted multilingual support

The ability to use as many languages as desired. A universal character set, such as Unicode, helps to provide unrestricted multilingual support because it supports a very large character repertoire, encompassing most modern languages of the world.

UTFE

A Unicode 3.0 UTF-8 Oracle database character set with 6-byte supplementary character support. It is used only on EBCDIC platforms.

UTF8

The UTF8 Oracle character set encodes characters in one, two, or three bytes. It is for ASCII-based platforms. The UTF8 character set supports Unicode 3.0. Although specific supplementary characters were not assigned code points in Unicode until version 3.1, the code point range was allocated for supplementary characters in Unicode 3.0. Supplementary characters are treated as two separate, user-defined characters that occupy 6 bytes.

UTF-8

The 8-bit encoding of Unicode. It is a variable-width encoding. One Unicode character can be 1 byte, 2 bytes, 3 bytes, or 4 bytes in UTF-8 encoding. Characters from the European scripts are represented in either 1 or 2 bytes. Characters from most Asian scripts are represented in 3 bytes. Supplementary characters are represented in 4 bytes.

UTF-16

The 16-bit encoding of Unicode. It is an extension of UCS-2 and supports the supplementary characters defined in Unicode 3.1 by using a pair of UCS-2 code points. One Unicode character can be 2 bytes or 4 bytes in UTF-16 encoding. Characters (including ASCII characters) from European scripts and most Asian scripts are represented in 2 bytes. Supplementary characters are represented in 4 bytes.

wide character

A fixed-width character format that is useful for extensive text processing because it allows data to be processed in consistent, fixed-width chunks. Wide characters are intended to support internal character processing.

Index