Oracle® Database

Globalization Support Guide

12*c* Release 1 (12.1)

E41669-07

August 2015

ORACLE®

Oracle Database Globalization Support Guide, 12*c* Release 1 (12.1)

E41669-07

Primary Author: Rajesh Bhatiya

Contributor: The Oracle Database 12*c* documentation is dedicated to Mark Townsend, who was an inspiration to all who worked on this release.

Contributors: Dan Chiba, Winson Chu, Claire Ho, Gary Hua, Simon Law, Geoff Lee, Peter Linsley, Qianrong Ma, Keni Matsuda, Meghna Mehta, Valarie Moore, Cathy Shea, Shige Takeda, Linus Tanaka, Makoto Tozawa, Barry Trute, Ying Wu, Peter Wallack, Chao Wang, Huaqing Wang, Sergiusz Wolicki, Simon Wong, Michael Yau, Jianping Yang, Qin Yu, Tim Yu, Weiran Zhang, Yan Zhu

Contents

3 Setting Up a Globalization Support Environment

4 Datetime Data Types and Time Zone Support

5 Linguistic Sorting and Matching

6 Supporting Multilingual Databases with Unicode

7 Programming with Unicode

8 Oracle Globalization Development Kit

9 SQL and PL/SQL Programming in a Global Environment

10 OCI Programming in a Global Environment

11 Character Set Migration

12 Customizing Locale Data

Preface

This book describes Oracle globalization support for Oracle Database. It explains how to set up a globalization support environment, choose and migrate a character set, customize locale data, do linguistic sorting, program in a global environment, and program with Unicode.

This preface contains these topics:

Intended Audience

Oracle 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 must be familiar with relational database concepts, basic Oracle Database concepts, and the operating system environment under which you are running Oracle.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or

visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documentation

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 *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Changes in This Release for Oracle Database Globalization Support Guide

This chapter contains:

Changes in Oracle Database 12c Release 1 (12.1)

The following are changes in *Oracle Database Globalization Support Guide* for Oracle Database 12*c* Release 1 (12.1).

New Features

- Support for Unicode 6.2, a major version of the Unicode Standard that supersedes all previous versions of the standard.
- Support for new locales.

  See Appendix A, "Locale Data".
- Support for the Unicode Collation Algorithm

  See Chapter 5, "Linguistic Sorting and Matching".
- The Database Migration Assistant for Unicode (DMU)

  An intuitive and user-friendly GUI product that helps you streamline the migration process through an interface that minimizes the manual workload and ensures that the migration tasks are carried out correctly and efficiently. It replaces the `CSSCAN` and `CSALTER` utilities as the supported method for migrating databases to Unicode.

  See "Migrating Character Data Using the Database Migration Assistant for Unicode" on page 11-6 for more details and *Oracle Database Migration Assistant for Unicode Guide* for more details.

Desupported Features

Some features previously described in this document (the `CSSCAN` and `CSALTER` utilities) are desupported in Oracle Database 12*c* Release 1 (12.1). See *Oracle Database Upgrade Guide* for a list of desupported features.

1

Overview of Globalization Support

This chapter provides an overview of globalization support for Oracle Database. This chapter discusses the following topics:

- Globalization Support Architecture
- Globalization Support Features

Globalization Support Architecture

The globalization support in Oracle Database 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 referred to globalization support capabilities as National Language Support (NLS) features. NLS is actually a subset of globalization support. NLS 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 Database globalization support is implemented with the Oracle NLS Runtime Library (NLSRTL). NLSRTL provides a comprehensive suite of language-independent functions that perform 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 run time.

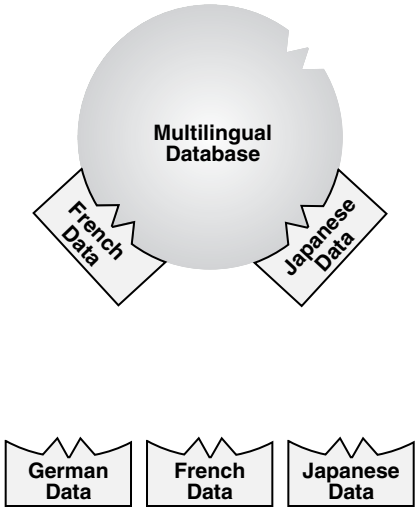
The locale-specific data is structured as independent sets of data for each locale that Oracle Database supports. The data for a particular locale can be loaded independently of other locale data.

The advantages of this design are as follows:

- You can manage memory consumption 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 how locale-specific data is loaded at run time. 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 the `$ORACLE_HOME/nls/data` directory. The `ORA_NLS10` environment variable should be defined only when you need to change the default directory location for the locale-specific data files, for example, when the system has multiple Oracle Database homes that share a single copy of the locale-specific data files.

A boot file is used to determine the availability of the NLS objects that can be loaded. Oracle Database 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"

Architecture to Support Multilingual Applications

Oracle Database enables 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 that is started on behalf of a client may run in the same or a different locale as other sessions, and can have the same or different language requirements specified.

Oracle Database has a set of session-independent NLS parameters that are specified when you create a database. Two of the parameters specify the database character set and the national character set, which is an alternative 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, such as language and territory, are used to evaluate and check constraints.

If the client session and the database server specify different character sets, then the 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 Database instance. For example, when SQL*Plus is started by the UNIX user who owns the Oracle Database 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-05      962.5
Urman                     07-MAR-06        975
Popp                      07-DIC-07      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 10, "OCI Programming in a Global Environment"
- Chapter 3, "Setting Up a Globalization Support Environment"

Using Unicode in a Multilingual Database

Unicode, the universal encoded character set, enables you to store information in any language by using a single character set. Unicode provides a unique code value for every character, regardless of the platform, program, or language. Oracle Corporation recommends using AL32UTF8 as the database character set. AL32UTF8 is the proper implementation of the UTF-8 encoding form of the Unicode Standard.

Unicode has the following advantages:

- Simplifies character set conversion and linguistic sort functions.

- Improves performance compared with native multibyte character sets.
- Supports the Unicode data type based on the Unicode standard.

To help you migrate to a Unicode environment, Oracle provides the Database Migration Assistant for Unicode (DMU). The DMU is an intuitive and user-friendly GUI that helps streamline the migration process through an interface that minimizes the workload and ensures that all migration issues are addressed, along with guaranteeing that the data conversion is carried out correctly and efficiently. The DMU offers many advantages over past methods of migrating data, some of which are:

- It guides you through the workflow.
- It offers suggestions for handling certain problems, such as failures during the cleansing of the data.
- It supports selective conversion of data.
- It offers progress monitoring.

See Also:

- Chapter 6, "Supporting Multilingual Databases with Unicode"
- Chapter 7, "Programming with Unicode"
- "Enabling Multilingual Support with Unicode Data Types" on page 6-8
- *Oracle Database Migration Assistant for Unicode Guide*

Globalization Support Features

This section provides an overview of the standard globalization features in Oracle Database:

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

Language Support

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

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

When this document uses the term **language support**, it refers to the additional language-dependent functionality, and not to the ability to store text of a specific language. For example, language support includes displaying dates or sorting text according to specific locales and cultural conventions. Additionally, for some supported languages, Oracle Database provides translated error messages and a translated user interface for the database utilities.

See Also:

- Chapter 3, "Setting Up a Globalization Support Environment"
- "Languages" on page A-1 for a complete list of Oracle Database language names and abbreviations
- "Translated Messages" on page A-4 for a list of languages into which Oracle Database messages are translated

Territory Support

Oracle 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. Setting different NLS parameters enables the database session to use different cultural settings. For example, you can set the euro (`EUR`) 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 Oracle Database

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"
- Chapter 4, "Datetime Data Types and Time Zone Support"
- *Oracle Database SQL Language 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"

Calendar Systems

Many different calendar systems are in use around the world. Oracle Database supports eight different calendar systems:

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

See Also:

- Chapter 3, "Setting Up a Globalization Support Environment"
- "Calendar Systems" on page A-24 for more information about supported calendars

Linguistic Sorting

Oracle Database 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 5, "Linguistic Sorting and Matching"

Character Set Support

Oracle Database 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

Oracle Database provides 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 2-9

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 12-36

Unicode Support

Unicode is an industry standard that enables text and symbols from all languages to be consistently represented and manipulated by computers. The latest version of the Unicode standard, as of August 2013, is 6.2.

Oracle Database has complied with the Unicode standard since Oracle 7. Subsequently, Oracle Database 10*g* release 2 supports Unicode 4.0. Oracle Database 11*g* release supports Unicode 5.0. Oracle Database 12*c* supports Unicode 6.2.

You can store Unicode characters in an Oracle database in two ways:

- You can create a Unicode database that enables you to store UTF-8 encoded characters as SQL `CHAR` data types.
- You can support multilingual data in specific columns by using Unicode data types. You can store Unicode characters into columns of the SQL `NCHAR` data types regardless of how the database character set has been defined. The `NCHAR` data type is an exclusively Unicode data type.

See Also: Chapter 6, "Supporting Multilingual Databases with Unicode"

2

Choosing a Character Set

This chapter explains how to choose a character set. The following topics are included:

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 the computer system interprets 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 discusses the following topics:

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

- Database performance
- Storage required for storing character data

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 a unique numeric code to each character in the character set. The numeric codes are called **code points** or **encoded values**. Table 2–1 shows examples of characters that have been assigned a hexadecimal code value in the ASCII character set.

Table 2–1 Encoded Characters in the ASCII Character Set

Character	Description	Hexadecimal Code Value
!	Exclamation Mark	21
-	Number Sign	23
-	Dollar Sign	24
-	Dollar Sign	24
-	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 to be used in the set
- The characters that are available to be used in the set (also known as the **character repertoire**)
- The scripts used for writing and the languages that they represent
- The code points or values assigned to each character
- The encoding scheme used to represent a specific character

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

> **See Also:** "Character Sets" on page A-6 for a complete list of character sets that are supported by Oracle Database

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 a group of languages. Writing systems can be classified into two categories:

- Phonetic Writing Systems

This section also includes the following topics:

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 multiple languages. 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 are encoded in more than one byte because the character repertoire has tens of thousands of characters.

Punctuation, Control Characters, Numbers, and Symbols

In addition to encoding the script of a language, other special characters must 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 Database 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 multiple languages. When character sets were first developed, 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` parameter properly is essential to proper data conversion. The character set that is specified by the `NLS_LANG` parameter should reflect the setting for the client operating system. Setting `NLS_LANG` correctly enables proper conversion from the client operating system character encoding to the database character set. When these settings are the same, Oracle Database assumes that the data being sent or received is encoded in the same character set as the database character set, so character set validation or conversion may not be performed. This can lead to corrupt data if conversions are necessary.

During conversion from one character set to another, Oracle Database expects client-side data to be encoded in the character set specified by the `NLS_LANG` 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 globalized 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 recommends that you use Unicode databases and data types.

See Also:

- Chapter 6, "Supporting Multilingual Databases with Unicode"
- *Oracle Database SQL Language Reference* for more information about the `CONVERT` SQL functions
- *Oracle Database SQL Language Reference* for more information about the `CHR` SQL functions
- "Displaying a Code Chart with the Oracle Locale Builder" on page 12-15

ASCII Encoding

Table 2–2 shows how the ASCII character set 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 code 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
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

As languages evolve to meet the needs of people around the world, new character sets are created to support the languages. 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)

Historically, character sets have provided restricted multilingual support, which has been limited to groups of languages based on similar scripts. More recently, universal character sets have emerged to enable greatly improved solutions for multilingual support. Unicode is one such universal character set that encompasses most major scripts of the modern world. As of version 6.2, the Unicode Standard encodes more than 110,000 characters.

See Also: Chapter 6, "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 efficient. 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 types:

- 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 shows the ISO 8859-1 8-bit encoding scheme.

Figure 2–1 ISO 8859-1 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, then 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 Database Character Sets

Oracle Database uses the following naming convention for its character set names:

```
<region><number of bits used to represent a character><standard character set name>[S|C]
```

The parts of the names that appear between angle brackets are concatenated. 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: Keep in mind that:

- You should 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.
- UTF8 and UTFE are exceptions to the naming convention.

Table 2–4 shows examples of Oracle Database character set names.

Table 2–4 Examples of Oracle Database Character Set Names

Oracle Database 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

Subsets and Supersets

When discussing character set conversion or character set compatibility between databases, Oracle documentation sometimes uses the terms *superset*, *subset*, *binary superset*, or *binary subset* to describe relationship between two character sets. The terms *subset* and *superset*, without the adjective "binary", pertain to character repertoires of two Oracle character sets, that is, to the sets of characters supported (encoded) by each of the character sets. By definition, character set A is a superset of character set B if A supports all characters that B supports. Character set B is a subset of character set A if A is a superset of B.

The terms *binary subset* and *binary superset* restrict the above subset-superset relationship by adding a condition on binary representation (binary codes) of characters of the two character sets. By definition, character set A is a binary superset of character set B if A supports all characters that B supports and all these characters have the same binary representation in A and B. Character set B is a binary subset of character set A if A is a binary superset of B.

When character set A is a binary superset of character set B, any text value encoded in B is at the same time valid in A without need for character set conversion. When A is a non-binary superset of B, a text value encoded in B can be represented in A without loss of data but may require character set conversion to transform the binary representation.

Oracle Database does not maintain a list of all subset-superset pairs but it does maintain a list of binary subset-superset pairs that it recognizes in various situations such as checking compatibility of a transportable tablespace or a pluggable database. Table A–11, " Binary Subset-Superset Pairs" and Table A–12, " Binary Supersets of US7ASCII" list the binary subset-superset pairs recognized by Oracle Database

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 point 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**.

Character semantics 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 length semantics of character data type columns, user-defined type attributes, and PL/SQL variables can be specified explicitly in their definitions with the `BYTE` or `CHAR` qualifier. This method of specifying the length semantics is recommended as it properly documents the expected semantics in creation DDL statements and makes the statements independent of any execution environment.

If a column, user-defined type attribute or PL/SQL variable definition contains neither the `BYTE` nor the `CHAR` qualifier, the length semantics associated with the column,

attribute, or variable is determined by the value of the session parameter NLS_LENGTH_SEMANTICS. If you create database objects with legacy scripts that are too large and complex to be updated to include explicit BYTE and/or CHAR qualifiers, execute an explicit ALTER SESSION SET NLS_LENGTH_SEMANTICS statement before running each of the scripts to assure the scripts create objects in the expected semantics.

The NLS_LENGTH_SEMANTICS initialization parameter determines the default value of the NLS_LENGTH_SEMANTICS session parameter for new sessions. Its default value is BYTE. For the sake of compatibility with existing application installation procedures, which may have been written before character length semantics was introduced into Oracle SQL, Oracle recommends that you leave this initialization parameter undefined or you set it to BYTE. Otherwise, created columns may be larger than expected, causing applications to malfunction or, in some cases, cause buffer overflows.

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 data types. The user cannot specify the CHAR or BYTE qualifier for NCHAR definitions.

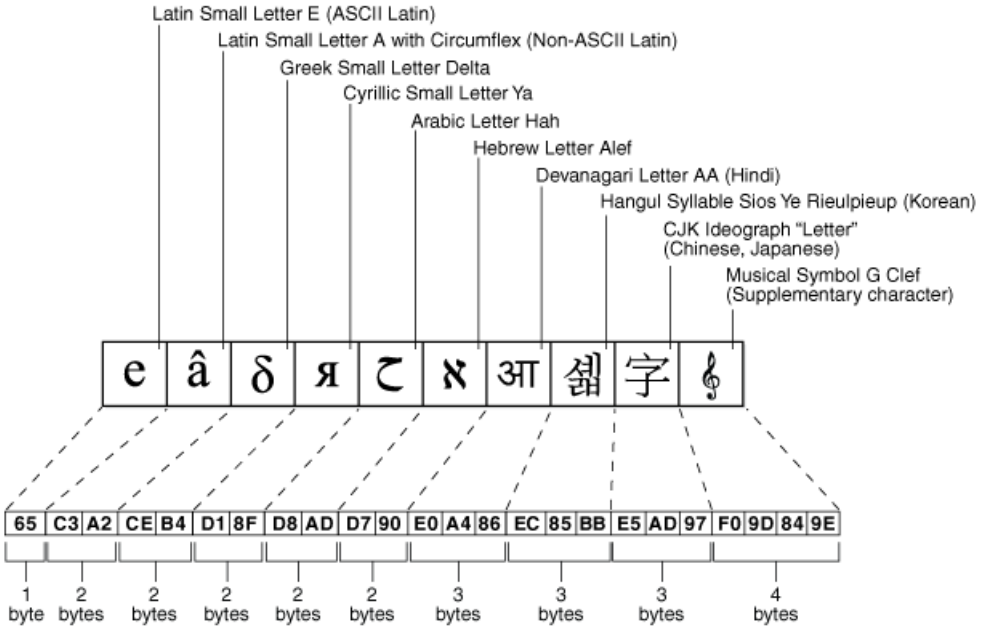
Consider the following example:

```
CREATE TABLE employees
( employee_id NUMBER(4)
, last_name NVARCHAR2(10)
, job_id NVARCHAR2(9)
, manager_id NUMBER(4)
, hire_date DATE
, salary NUMBER(7,2)
, department_id NUMBER(2)
) ;
```

last_name can hold up to 10 Unicode code points, independent of whether the NCHAR character set is AL16UTF16 or UTF8. When the NCHAR character set is AL16UTF16, these stored 10 code points may occupy up to 20 bytes. When the NCHAR character set is UTF8, they may occupy up to 30 bytes.

Figure 2–2 shows the number of bytes needed to store different kinds of characters in the UTF-8 character set. The ASCII character requires one byte, the non-ASCII Latin, Greek, Cyrillic, Arabic, and Hebrew characters require two bytes, the Asian characters require 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 9-4 for more information about the `SUBSTR` and `SUBSTRB` functions
- "Length Semantics" on page 3-32 for more information about the `NLS_LENGTH_SEMANTICS` initialization parameter
- Chapter 6, "Supporting Multilingual Databases with Unicode" for more information about Unicode and the `NCHAR` data type
- *Oracle Database SQL Language Reference* for more information about the `SUBSTRB` and `SUBSTR` functions and the `BYTE` and `CHAR` qualifiers for character data types

Choosing an Oracle Database Character Set

Oracle Database uses the database character set for:

- Data stored in SQL `CHAR` data types (`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` data type 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` data type columns (`NCHAR`, `NCLOB`, and `NVARCHAR2`) use the national 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 Database character set for the database:

- What languages does the database need to support now?
- What languages does 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 Database character sets are listed in "Character Sets" on page A-6. 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:

See Also:

- "UCS-2 Encoding Form" on page 6-4
- "Choosing a National Character Set" on page 2-15
- "Changing the Character Set After Database Creation" on page 2-18
- 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. You should generally select the Unicode character set AL32UTF8, because it supports most languages of the world.

Client Operating System and Application Compatibility

The database character set is independent of the operating system because Oracle Database 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 an application in 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:

- Potential data loss
- Increased overhead

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, then 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 11, "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 2–5 lists the restrictions on the character sets that can be used to express names.

Table 2–5 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	-
Database link names	Yes	No	-
Database names	Yes	No	-
File names (data file, 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 2–7 on page 2-16.

Database Character Set Statement of Direction

A list of character sets has been compiled in Table A–4, " Recommended ASCII Database Character Sets" and Table A–5, " Recommended EBCDIC Database Character Sets" that Oracle strongly recommends for usage as the database character set. Other Oracle-supported character sets that do not appear on this list can continue to be used in Oracle Database 12*c*, but may be desupported in a future release. Starting with Oracle Database 11*g* Release 1, the choice for the database character set is limited to this list of recommended character sets in common installation paths of Oracle Universal Installer and Oracle Database Configuration Assistant. Customers are still able to create new databases using custom installation paths and migrate their existing databases even if the character set is not on the recommended list. However, Oracle suggests that customers migrate to a recommended character set as soon as possible. At the top of the list of character sets that Oracle recommends for all new system deployment, is the Unicode character set AL32UTF8.

Choosing Unicode as a Database Character Set

Oracle recommends using Unicode for all new system deployments. Migrating legacy systems to Unicode is also recommended. Deploying your systems today in Unicode offers many advantages in usability, compatibility, and extensibility. Oracle Database enables you to deploy high-performing systems faster and more easily while utilizing the advantages of Unicode. Even if you do not need to support multilingual data today, nor have any requirement for Unicode, it is still likely to be the best choice for a new system in the long run and will ultimately save you time and money as well as give you competitive advantages in the long term.

> **See Also:** Chapter 6, "Supporting Multilingual Databases with Unicode"

Choosing a National Character Set

The term **national character set** refers to an alternative character set that enables you to store Unicode character data in a database that does not have a Unicode database character set. Another reason for choosing a national character set is that the properties of a different character encoding scheme may be more desirable for extensive character processing operations.

SQL `NCHAR`, `NVARCHAR2`, and `NCLOB` data types support Unicode data only. You can use either the UTF8 or the AL16UTF16 character set. The default is AL16UTF16.

Oracle recommends using SQL `CHAR`, `VARCHAR2`, and `CLOB` data types in AL32UTF8 database to store Unicode character data. Use of SQL `NCHAR`, `NVARCHAR2`, and `NCLOB` should be considered only if you must use a database whose database character set is not AL32UTF8.

> **See Also:** Chapter 6, "Supporting Multilingual Databases with Unicode"

Summary of Supported Data Types

Table 2–6 lists the data types that are supported for different encoding schemes.

Table 2–6 SQL Data Types Supported for Encoding Schemes

Data Type	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 2–7 lists the SQL data types that are supported for abstract data types.

Table 2–7 Abstract Data Type Support for SQL Data Types

Abstract Data Type	CHAR	NCHAR	BLOB	CLOB	NCLOB
Object	Yes	Yes	Yes	Yes	Yes
Collection	Yes	Yes	Yes	Yes	Yes

You can create an abstract data type 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:** *Oracle Database Object-Relational Developer's Guide* for more information about objects and collections

Choosing a Database Character Set for a Multitenant Container Database

In Oracle Database 12*c*, all containers of a multitenant container database (CDB) must have the same database character set and the same national character set. That is, all pluggable databases (PDBs) in a CDB must have the same character set as the CDB's root container. The character sets of the root container are considered the character sets of the whole CDB.

You can consolidate multiple databases by plugging them into a single CDB. These databases can be traditional, non-CDBs or PDBs unplugged from other CDBs. Three situations may happen depending on the database character set of a database being plugged in, that is, a new PDB:

- The character set is the same as the database character set of the CDB. In this case, the plugging operation succeeds (as far as the database character set is concerned).
- The character set is plug-in compatible with the database character set of the CDB. Plug-in compatible means that it is a binary subset of the CDB's character set and both are single-byte or both are multibyte. In this case, the database character set of the new PDB is automatically changed to the database character set of the CDB at the first open and the plugging operation succeeds.
- The character set is not plug-in compatible with the database character set of the CDB. In this case, the new PDB can only be opened in restricted mode for administrative tasks and cannot be used for production. Unless you have a tool that can migrate the database character set of the new PDB to the character set of the CDB, the new PDB is unusable.

> **See Also:**
>
> - "Subsets and Supersets" on page 2-8 for a discussion of what a binary subset of a character set is

When you plug a new PDB, two situations may happen depending on the national character set of the new PDB:

- The character set is the same as the national character set of the CDB. In this case, the plugging operation succeeds (as far as the national character set is concerned).
- The character set is not the same as the national character set of the CDB. In this case, the new PDB can only be opened in restricted mode for administrative tasks and cannot be used for production. Unless you have a tool that can migrate the

national character set of the new PDB to the character set of the CDB, the new PDB is unusable.

Because of these restrictions, you should consider the character sets of databases that you want to plug into a given CDB when selecting character sets for this CDB. Oracle recommends the following:

- For all new deployments and if all PDBs are created empty, Oracle strongly recommends AL32UTF8 for the CDB database character set and AL16UTF6 for the CDB national character set.
- If you can migrate your existing databases to AL32UTF8 before consolidation, Oracle recommends that you do so and consolidate into one or more AL32UTF8 CDBs, depending on your needs. You can use Oracle Database Migration Assistant for Unicode to migrate a non-CDB to AL32UTF8.
- If you cannot migrate your existing databases prior to consolidation, then you have to partition them into sets with plug-in compatible database character sets and plug each set into a separate CDB with the appropriate superset character set. For example, if you have US7ASCII, WE8ISO8859P1, WE8MSWIN1252, WE8ISO8859P15, JA16SJISTILDE, JA16EUC, KO16KSC5601, KO16MSWIN949, UTF8 and AL32UTF8 databases that you want to consolidate, you need to consolidate as follows:
  - US7ASCII, WE8ISO8859P1, and WE8MSWIN1252 into a WE8MSWIN1252 CDB
  - WE8ISO8859P15 into a WE8ISO8859P15 CDB
  - JA16SJISTILDE into a JA16SJISTILDE CDB
  - JA16EUC into a JA16EUC CDB
  - KO16KSC5601, KO16MSWIN949 into a KO16MSWIN949 CDB
  - UTF8 and AL32UTF8 into an AL32UTF8 CDB

Note: Because UTF8 and AL32UTF8 have different maximum character widths (three versus four bytes per character), the automatic change of UTF8 to AL32UTF8 during plug-in operation will change implicit maximum byte lengths of columns with character length semantics. This change may fail if there are functional indexes, virtual columns, bitmap join indexes, domain indexes, partitioning keys, sub-partitioning key, or cluster keys defined on those columns. The operation may also fail, if a character length semantics column is part of an index key and the index key exceeds the size limit after the character set change (around 70% of the index block size). You must make sure that all offending objects are removed from the database before the database is plugged in. You can recreate the objects after the operation.

See Also:

- *Oracle Database Concepts* and *Oracle Database Administrator's Guide* for more information about CDBs and PDBs
- *Oracle Database Migration Assistant for Unicode Guide* for more information about migrating the database character set to Unicode

Changing the Character Set After Database Creation

You may want to change the database character set after the database has been created. For example, you may find that the number of languages that must be supported in your database has increased, and you therefore want to migrate to Unicode's AL32UTF8.

As character type data in the database must be converted to Unicode, in most cases, you will encounter challenges when you change the database character set to AL32UTF8. For example, `CHAR` and `VARCHAR2` column data may exceed the declared column length. Character data may be lost when it is converted to Unicode if it contains invalid characters.

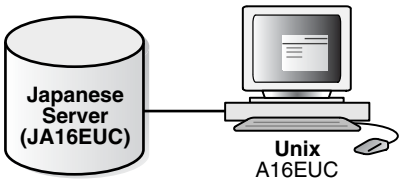
Before changing the database character set, it is important to identify all problems and carefully plan the data migration. Oracle recommends using the Database Migration Assistant for Unicode to change the database character set to AL32UTF8.

> **See Also:** *Oracle Database Migration Assistant for Unicode Guide* for more information about how to change character sets

Monolingual Database Scenario

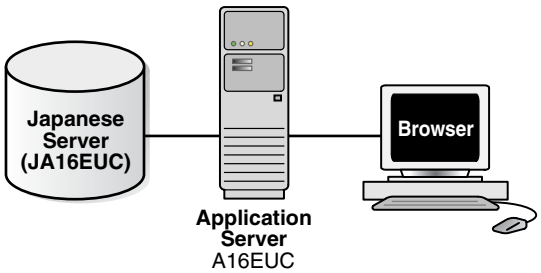
The simplest example of a database configuration is a client and a server that run in the same language environment and use the same character set. This monolingual scenario has the advantage of fast response because the overhead associated with character set conversion is avoided. Figure 2–3 shows a database server and a client that use the same character set. The Japanese client and the server both use the JA16EUC character set.

Figure 2–3 Monolingual Database Scenario



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

Figure 2–4 Multitier Monolingual Database Scenario

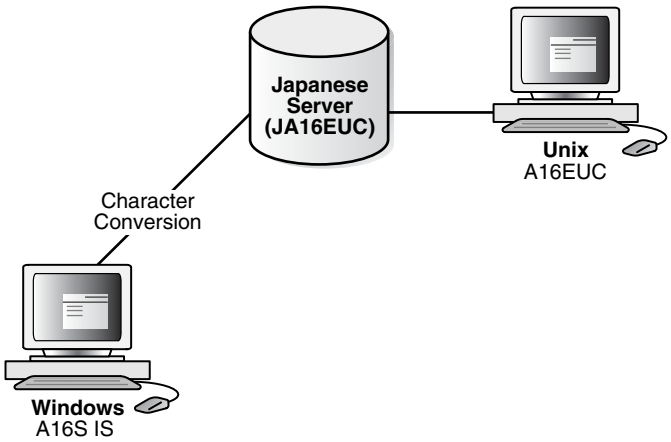


Character Set Conversion in a Monolingual Scenario

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 through 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

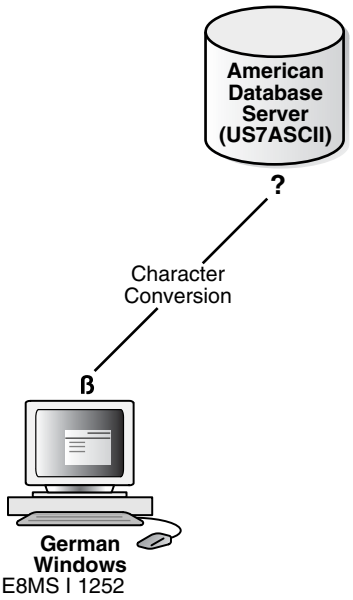


When a target character set does not contain all of the characters in the source data, replacement characters are used. If, for example, a server uses US7ASCII and a German client uses WE8ISO8859P1, then the German character ß is replaced with ? and ä is replaced with a.

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. 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.

Figure 2–6 Data Loss During Character Conversion



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 Scenario

If you need multilingual support, then use Unicode AL32UTF8 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.

Oracle Database provides support for UTF-8 as a database character set and both UTF-8 and UTF-16 as national character sets.

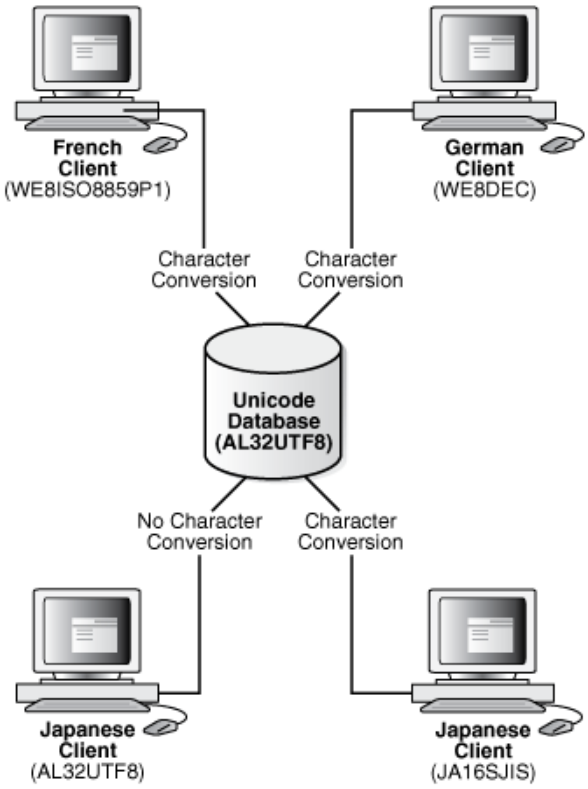
Character set conversion between a UTF-8 database and any single-byte character set introduces very little overhead.

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 one 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 2–7 shows a server that uses the AL32UTF8 Oracle Database character set that is based on the Unicode UTF-8 character set.

Figure 2–7 Multilingual Support Scenario in a Client/Server Configuration



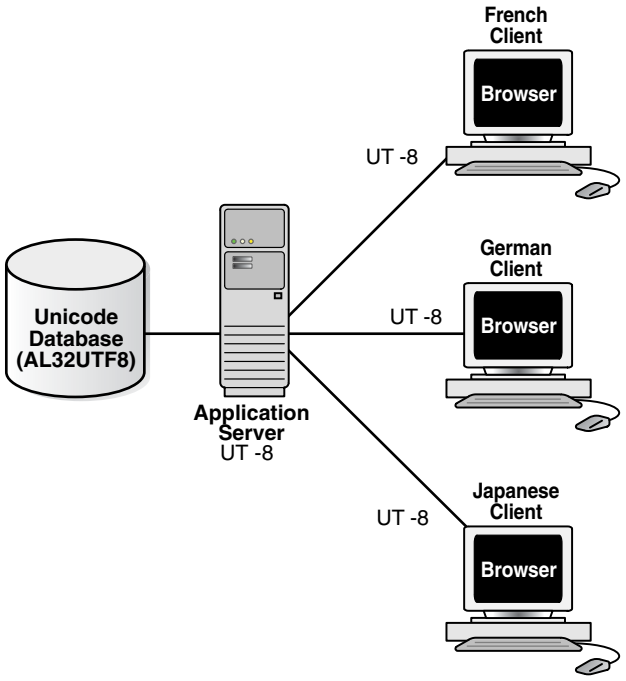
There are four clients:

- A French client that uses the WE8ISO8859P1 Oracle Database character set
- A German client that uses the WE8DEC character set
- A Japanese client that uses the AL32UTF8 character set
- A Japanese client that used the JA16SJIS character set

Character conversion takes place between each client and the server except for the AL32UTF8 client, 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, then all of the Japanese characters in the data are lost during the character set conversion.

Figure 2–8 shows a Unicode solution for a multitier configuration.

Figure 2–8 Multitier 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 6, "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 (National Language Support) 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 environment variables, which may be platform-dependent, 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

  You can use NLS parameters that are set in an `ALTER SESSION` statement 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: *Oracle Database SQL Language Reference* for more information about the `ALTER SESSION` statement

- In SQL functions

  You can use NLS parameters explicitly to hardcode NLS behavior within a SQL function. This practice overrides 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: *Oracle Database SQL Language 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.

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 available NLS parameters. Because the SQL function NLS parameters can be specified only with specific functions, the table does not show the SQL function scope. This table shows the following values for Scope:

- I = Initialization Parameter File
- E = Environment Variable
- A = ALTER SESSION

Table 3–2 NLS Parameters

Parameter	Description	Default	Scope
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

Table 3–2 (Cont.) NLS Parameters

Parameter	Description	Default	Scope
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
NLS_LANG **See Also:** "Choosing a Locale with the NLS_LANG Environment Variable" on page 3-3	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, E, 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 for Oracle Database software. It sets the language and territory used by the client application and the database server. It also sets the client's character set, which is the character set for data entered or displayed by a client program.

NLS_LANG is set as an environment variable on UNIX platforms. NLS_LANG is set in the registry on Windows platforms.

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 Universal Installer does not populate NLS_LANG, then its value by default is AMERICAN_AMERICA.US7ASCII. The language is AMERICAN, the territory is AMERICA, and the character set is US7ASCII. The values in NLS_LANG and other NLS parameters are case-insensitive.

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 Database 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 the Oracle Database character set that corresponds to the user's terminal character set or the OS character set). 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 that is not specified uses its default value. If you specify territory or character set, then you *must* include the preceding delimiter [underscore (_) for territory, period (.) for character set]. Otherwise, the value is parsed as a language name.

For example, to set only the territory portion of NLS_LANG, use the following format: NLS_LANG=_JAPAN

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

```
NLS_LANG = AMERICAN_AMERICA.WE8MSWIN1252

NLS_LANG = FRENCH_CANADA.WE8ISO8859P1

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.WE8ISO8859P1
```

Because the WE8ISO8859P1 character set does not support any Japanese characters, you cannot store or display 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
- Locale Variants

See Also:

- Appendix A, "Locale Data" for a complete list of supported languages, territories, and character sets
- Your operating system documentation for information about additional globalization settings that may be necessary for your platform

Specifying the Value of NLS_LANG

In a UNIX operating system C-shell session, you can specify the value of NLS_LANG by entering a statement similar to the following example:

```
% setenv NLS_LANG FRENCH_FRANCE.WE8ISO8859P1
```

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 Database 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 output:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
...
Sciarra                   30-SEP-05      962.5
Urman                     07-MAR-06        975
Popp                      07-DEC-07      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 Database 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_DATE     SALARY
------------------------- ---------  ---------
...
Sciarra                   30/09/05       962,5
Urman                     07/03/06         975
Popp                      07/12/07       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 Database 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_TERRITORY, and other NLS instance parameters from the initialization parameter file. You can modify these parameters and restart the instance to change the defaults.

You might want to modify 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 3-14
- *Oracle Database SQL Language Reference*

Locale Variants

Before Oracle Database 10*g*, Oracle defined language and territory definitions separately. This resulted in the definition of a territory being independent of the language setting of the user. Since Oracle Database 10*g*, some territories can have different date, time, number, and monetary formats based on the language setting of a user. This type of language-dependent territory definition is called a locale variant.

For the variant to work properly, both NLS_TERRITORY and NLS_LANGUAGE must be specified.

Table 3–3 shows the territories that have been enhanced to support variations.

Table 3–3 Oracle Database Locale Variants

Oracle Database Territory	Oracle Database Language
BELGIUM	DUTCH
BELGIUM	FRENCH
BELGIUM	GERMAN
CANADA	FRENCH
CANADA	ENGLISH
DJIBOUTI	FRENCH
DJIBOUTI	ARABIC
FINLAND	FINLAND
FINLAND	SWEDISH
HONG KONG	TRADITIONAL CHINESE
HONG KONG	ENGLISH
INDIA	ENGLISH
INDIA	ASSAMESE
INDIA	BANGLA
INDIA	GUJARATI
INDIA	HINDI
INDIA	KANNADA
INDIA	MALAYALAM
INDIA	MARATHI
INDIA	ORIYA
INDIA	PUNJABI
INDIA	TAMIL
INDIA	TELUGU
LUXEMBOURG	GERMAN
LUXEMBOURG	FRENCH
SINGAPORE	ENGLISH
SINGAPORE	MALAY
SINGAPORE	SIMPLIFIED CHINESE
SINGAPORE	TAMIL
SWITZERLAND	GERMAN
SWITZERLAND	FRENCH
SWITZERLAND	ITALIAN

Should the NLS_LANG Setting Match the Database Character Set?

The NLS_LANG character set should reflect the setting of the operating system character set of the client. For example, if the database character set is AL32UTF8 and the client is running on a Windows operating system, then you should not set AL32UTF8 as the

client character set in the NLS_LANG parameter because there are no UTF-8 WIN32 clients. Instead, the NLS_LANG setting should reflect the code page of the client. For example, on an English Windows client, the code page is 1252. An appropriate setting for NLS_LANG is AMERICAN_AMERICA.WE8MSWIN1252.

Setting NLS_LANG correctly enables proper conversion from the client operating system character set to the database character set. When these settings are the same, Oracle Database assumes that the data being sent or received is encoded in the same character set as the database character set, so character set validation or conversion may not be 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:** *Oracle Database Installation Guide for Microsoft Windows* for more information about commonly used values of the NLS_LANG parameter in Windows

Character Set Parameter

Oracle provides an environment variable, NLS_OS_CHARSET, for handling the situation where the client OS character set is different from the Oracle NLS client character set.

NLS_OS_CHARSET Environment Variable

The NLS_OS_CHARSET environment variable should be set on Oracle client installations if the client OS character set is different from the Oracle NLS client character set specified by the NLS_LANG environment variable. The client OS character set is the character set used to represent characters in the OS fields like machine name, program executable name and logged on user name. On UNIX platforms, this is usually the character set specified in the LANG environment variable or the LC_ALL environment variable. An example of setting NLS_OS_CHARSET would be if the locale charset specified in LANG or LC_ALL in a Linux client could be zh_CN (simplified Chinese) and the Oracle client application charset specified in NLS_LANG could be UTF8. In this case, the NLS_OS_CHARSET variable must be set to the equivalent Oracle charset ZHT16GBK.

The NLS_OS_CHARSET environment variable must be set to the Oracle character set name corresponding to the client OS character set.

If NLS_LANG corresponds to the OS character set, NLS_OS_CHARSET does not need to be set. NLS_OS_CHARSET does not need to be set and is ignored on Windows platforms.

NLS Database Parameters

When a new database is created during the execution of the CREATE DATABASE statement, the NLS-related database configuration 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 for the database. The values are stored in the database.

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: *Oracle Database Reference*

OCINlsGetInfo() Function

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

See Also: "Getting Locale Information in OCI" on page 10-2 for the description of OCINlsGetInfo()

Language and Territory Parameters

This section contains information about the following parameters:

- NLS_LANGUAGE
- NLS_TERRITORY

NLS_LANGUAGE

Property	Description
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
```

Consider the following server message:

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

When the language is French, the server message appears as follows:

```
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 file name convention:

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

For example, the file containing the server messages in French is called oraf.msb, because ORA is the product ID (<product_id>) and F is the language abbreviation (<language_abbrev>) for French. The *product_name* is rdbms, so it is in the $ORACLE_HOME/rdbms/mesg directory.

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

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 is then converted to the client character set if the client character set is different from the database character set. Hence, messages are displayed correctly at the user's terminal, subject to the limitations of character set conversion.

The language-specific binary message files that are actually installed depend on the languages that the user specifies during product installation. Only the English binary message file and the language-specific binary message files specified by the user are installed.

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 Database 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_LANGUAGE 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 output:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
...
Sciarra                   30-SET-05      962.5
Urman                     07-MAR-06        975
Popp                      07-DIC-07      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 3-14 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 output:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
...
Sciarra                   30-SEP-05      962.5
Urman                     07-MRZ-06        975
Popp                      07-DEZ-07      862.5
...
```

Note that the language of the month abbreviations has changed.

NLS_TERRITORY

Property	Description
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 output:

```
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 3-14 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 results similar to the following output:

```
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 and default values for specific formatting parameters 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 results similar to the following output:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
...
Sciarra                   30-SET-05      962,5
Urman                     07-MAR-06        975
Popp                      07-DIC-07      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 results similar to the following output:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
...
Sciarra                   30.SEP.05      962.5
Urman                     07.MRZ.06        975
Popp                      07.DEZ.07      862.5
...
```

Note that the language of the month abbreviations is German and the decimal character is a period.

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, then 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 Database 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 3–4.

Table 3–4 Date Formats

Country	Description	Example
Estonia	dd.mm.yyyy	28.02.2003
Germany	dd-mm-rr	28-02-03
Japan	rr-mm-dd	03-02-28
UK	dd-mon-rr	28-Feb-03
US	dd-mon-rr	28-Feb-03

This section includes the following parameters:

- NLS_DATE_FORMAT
- NLS_DATE_LANGUAGE

NLS_DATE_FORMAT

Property	Description
Parameter type	String
Parameter scope	Initialization parameter, environment variable, and ALTER SESSION
Default value	Derived from NLS_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. 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 when double quotes are included in the date format, the entire value must be enclosed by single quotes. For example:

```
NLS_DATE_FORMAT = '"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 14 bytes internally because there are three format elements (month, day, and year), two 3-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.

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:** *Oracle Database SQL Language 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:** *Oracle Database SQL Language Reference* for more information about partitioning tables and indexes and using TO_DATE

NLS_DATE_LANGUAGE

Property	Description
Parameter type	String
Parameter scope	Initialization parameter, environment variable, ALTER SESSION, and SQL functions
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

As an example of how to use NLS_DATE_LANGUAGE, set the date language to French:

```
ALTER SESSION SET NLS_DATE_LANGUAGE = FRENCH;
```

Enter a SELECT statement:

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

You should see results similar to the following output:

```
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 results similar to the following output:

```
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 results similar to the following output:

```
TO_CHAR(SYSDATE,'DY:DDMO
------------------------
Ve:07 Déc. 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-Févr.-1997');
```

See Also: *Oracle Database SQL Language Reference*

Time Formats

Different time formats are shown in Table 3–5.

Table 3–5 Time Formats

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

This section contains information about the following parameters:

- NLS_TIMESTAMP_FORMAT
- NLS_TIMESTAMP_TZ_FORMAT

See Also: Chapter 4, "Datetime Data Types and Time Zone Support"

NLS_TIMESTAMP_FORMAT

Property	Description
Parameter type	String
Parameter scope	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 date format for the TIMESTAMP and TIMESTAMP WITH LOCAL TIME ZONE data types. The following example shows a value for NLS_TIMESTAMP_FORMAT:

```
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 results similar to the following output:

```
TO_TIMESTAMP('11-NOV-200001:00:00.336','DD-MON-YYYYHH:MI:SS.FF')
---------------------------------------------------------------------------
2000-11-11 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:** *Oracle Database SQL Language Reference* for more information about the TO_TIMESTAMP function and the ALTER SESSION statement

NLS_TIMESTAMP_TZ_FORMAT

Property	Description
Parameter type	String
Parameter scope	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 date format for the TIMESTAMP and TIMESTAMP WITH LOCAL TIME ZONE data types. 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 results similar to the following output:

```
TO_TIMESTAMP_TZ('2000-08-20,05:00:00.55AMERICA/LOS_ANGELES','YYYY-MM-DDHH:M
---------------------------------------------------------------------------
2000-08-20 05:00:00.550000000 -07:00
```

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:

- *Oracle Database SQL Language Reference* for more information about the TO_TIMESTAMP_TZ function and the ALTER SESSION statement
- "Choosing a Time Zone File" on page 4-15 for more information about time zones

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 3–6.

Table 3–6 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 3-12

First Calendar Week of the Year

Some countries use week numbers for scheduling, planning, and bookkeeping. Oracle Database 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 Database provides the IW date format element. It returns the ISO week number.

Table 3–7 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 3–7 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 3–8 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 3–8 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
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 3-12

Number of Days and Months in a Year

Oracle Database 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
- Ethiopian—has 12 months of 30 days each, then a 13th month that is either five or six days (leap year). The sixth day of the 13th month is added every four years.

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

See Also: "NLS_CALENDAR" on page 3-22

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

Property	Description
Parameter type	String
Parameter scope	Initialization parameter, environment variable, `ALTER SESSION`, and SQL functions
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 Database uses.

`NLS_CALENDAR` can have one of the following values:

- Arabic Hijrah
- English Hijrah
- Ethiopian
- 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='English Hijrah'

Set NLS_CALENDAR to English Hijrah.

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

Enter a SELECT statement to display SYSDATE:

```
SELECT SYSDATE FROM DUAL;
```

You should see results similar to the following output:

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

Numeric and List Parameters

This section includes the following topics:

- Numeric Formats
- NLS_NUMERIC_CHARACTERS
- NLS_LIST_SEPARATOR

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 3–9.

Table 3–9 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 3-12

NLS_NUMERIC_CHARACTERS

Property	Description
Parameter type	String
Parameter scope	Initialization parameter, environment variable, ALTER SESSION, and SQL functions
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 character 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.

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 = ",.";
```

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 character 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.

The following SELECT statement formats the number 4000 with the decimal character and group separator specified in the ALTER SESSION statement:

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

You should see results similar to the following output:

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

> **See Also:** *Oracle Database SQL Language Reference* for more information about the ALTER SESSION statement

NLS_LIST_SEPARATOR

Property	Description
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 (usually , or . or ; or :). Its default value is derived from the value of NLS_TERRITORY. For example, a list of numbers from 1 to 5 can be expressed as 1,2,3,4,5 or 1.2.3.4.5 or 1;2;3;4;5 or 1:2:3:4:5.

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 (.).

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 3–10.

Table 3–10 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

Property	Description
Parameter type	String

Property	Description
Parameter scope	Initialization parameter, environment variable, ALTER SESSION, and SQL functions
Default value	Derived from NLS_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 order entry schema:

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

Enter a SELECT statement similar to the following example:

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

You should see results similar to the following output:

```
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:** *Oracle Database SQL Language Reference* for more information about the ALTER SESSION statement

NLS_ISO_CURRENCY

Property	Description
Parameter type	String
Parameter scope	Initialization parameter, environment variable, ALTER SESSION, and SQL functions
Default value	Derived from NLS_TERRITORY
Range of values	Any valid string

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. The ISO currency symbol for the Australian dollar is AUD.

More ISO currency symbols are shown in Table 3–11.

Table 3–11 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 results similar to the following output:

```
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:** *Oracle Database SQL Language Reference* for more information about the ALTER SESSION statement

NLS_DUAL_CURRENCY

Property	Description
Parameter type	String
Parameter scope	Initialization parameter, environmental variable, ALTER SESSION, and SQL functions
Default value	Derived from NLS_TERRITORY
Range of values	Any valid symbol

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. See Table A–8, " Character Sets that Support the Euro Symbol" for the character sets that support the euro symbol.

Oracle Database Support for the Euro

Twelve members of the European Monetary Union (EMU) have adopted the euro as their currency. 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 Database support for the euro was provided in Oracle Database 8*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 Oracle Database 9*i* Release 2, the value of NLS_ISO_CURRENCY results in the ISO currency symbol being set to EUR for EMU member countries who use the euro. For example, suppose NLS_ISO_CURRENCY is set to FRANCE. Enter the following SELECT statement:

```
SELECT TO_CHAR(order_total, 'C099G999D99') "TOTAL" FROM orders
   WHERE customer_id=116;
```

You should see results similar to the following output:

```
TOTAL
-------
EUR006,394.80
EUR011,097.40
EUR014,685.80
EUR000,129.00
```

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

Property	Description
Parameter type	String
Parameter scope	Environment variable
Default value	Derived from NLS_TERRITORY
Range of values	Any valid character

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

Property	Description
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

Property	Description
Parameter type	String
Parameter scope	Environment variable
Default value	Derived from NLS_TERRITORY
Range of values	Any string, maximum or 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

See Also: Chapter 5, "Linguistic Sorting and Matching"

NLS_SORT

Property	Description
Parameter type	String
Parameter scope	Initialization parameter, environment variable, `ALTER SESSION`, and SQL functions
Default value	Derived from `NLS_LANGUAGE`
Range of values	`BINARY` or any valid linguistic sort name

`NLS_SORT` specifies the type of sort for character data. It overrides the default value that is derived from `NLS_LANGUAGE`.

`NLS_SORT` contains either of the following values:

```
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 German linguistic sort sequence, set `NLS_SORT` as follows:

```
NLS_SORT = German
```

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`, then 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:

- Chapter 5, "Linguistic Sorting and Matching"
- *Oracle Database SQL Language Reference* for more information about the ALTER SESSION statement
- "Linguistic Sorts" on page A-20 for a list of linguistic sort names

NLS_COMP

Property	Description
Parameter type	String
Parameter scope	Initialization parameter, environment variable, and ALTER SESSION
Default value	BINARY
Range of values	BINARY , LINGUISTIC, or ANSI

The value of NLS_COMP affects the comparison behavior of SQL operations.

You can use NLS_COMP to avoid the cumbersome process of using the NLSSORT function in SQL statements when you want to perform a linguistic comparison instead of a binary comparison. When NLS_COMP is set to LINGUISTIC, SQL operations perform a linguistic comparison based on the value of NLS_SORT. A setting of ANSI is for backward compatibility; in general, you should set NLS_COMP to LINGUISTIC when you want to perform a linguistic comparison.

Set NLS_COMP to LINGUISTIC as follows:

```
ALTER SESSION SET NLS_COMP = LINGUISTIC;
```

When NLS_COMP is set to LINGUISTIC, 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 Collation" on page 5-3
- "Using Linguistic Indexes" on page 5-23

Character Set Conversion Parameter

This section includes the following topic:

- NLS_NCHAR_CONV_EXCP

NLS_NCHAR_CONV_EXCP

Property	Description
Parameter type	String
Parameter scope	Initialization parameter and ALTER SESSION
Default value	FALSE

Property	Description
Range of values	TRUE or FALSE

NLS_NCHAR_CONV_EXCP determines whether an error is reported when there is data loss during an implicit or explicit character type conversion between NCHAR/NVARCHAR data and CHAR/VARCHAR2 data. The default value results in no error being reported.

> **See Also:** Chapter 11, "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

Property	Description
Parameter type	String
Parameter scope	Environment variable, initialization parameter, and ALTER SESSION
Default value	BYTE
Range of values	BYTE or CHAR

By default, the character data types 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 is 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 created in the SYS schema. The data dictionary always uses byte semantics. Tables owned by SYS always use byte semantics if the length qualifier BYTE or CHAR is not specified in the table creation DDL.

Note that if the NLS_LENGTH_SEMANTICS environment variable is not set on the client, then the client session defaults to the value for NLS_LENGTH_SEMANTICS on the database server. This enables all client sessions on the network to have the same NLS_LENGTH_SEMANTICS behavior. Setting the environment variable on an individual client enables the server initialization parameter to be overridden for that client.

Note that if the NLS_LENGTH_SEMANTICS environment variable is not set on the client or the client connects through the Oracle JDBC Thin driver, then the client session defaults to the value for the NLS_LENGTH_SEMANTICS initialization parameter of the instance to which the client connects. For compatibility reasons, Oracle recommends that this parameter be left undefined or set to BYTE.

Caution: Oracle strongly recommends that you do NOT set the NLS_LENGTH_SEMANTICS parameter to CHAR in the instance or server parameter file. This may cause many existing installation scripts to unexpectedly create columns with character length semantics, resulting in run-time errors, including buffer overflows.

See Also: "Length Semantics" on page 2-9

4

Datetime Data Types and Time Zone Support

This chapter includes the following topics:

Overview of Datetime and Interval Data Types and Time Zone Support

Businesses conduct transactions across different time zones. Oracle Database datetime and interval data types and time zone support make it possible to store consistent information about the time of events and transactions.

> **Note:** This chapter describes Oracle Database datetime and interval data types. It does not attempt to describe ANSI data types or other kinds of data types unless noted.

Datetime and Interval Data Types

The **datetime data types** are `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `TIMESTAMP WITH LOCAL TIME ZONE`. Values of datetime data types are sometimes called **datetimes**.

The **interval data types** are `INTERVAL YEAR TO MONTH` and `INTERVAL DAY TO SECOND`. Values of interval data types are sometimes called **intervals**.

Both datetimes and intervals are made up of fields. The values of these fields determine the value of the data type. The fields that apply to all Oracle Database datetime and interval data types are:

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND

TIMESTAMP WITH TIME ZONE also includes these fields:

- TIMEZONE_HOUR
- TIMEZONE_MINUTE
- TIMEZONE_REGION
- TIMEZONE_ABBR

TIMESTAMP WITH LOCAL TIME ZONE does not store time zone information internally, but you can see local time zone information in SQL output if the TZH:TZM or TZR TZD format elements are specified.

The following sections describe the datetime data types and interval data types in more detail:

- Datetime Data Types
- Interval Data Types

See Also:

- *Oracle Database SQL Language Reference* for the valid values of the datetime and interval fields
- *Oracle Database SQL Language Reference* for information about format elements

Datetime Data Types

This section includes the following topics:

- DATE Data Type
- TIMESTAMP Data Type
- TIMESTAMP WITH TIME ZONE Data Type
- TIMESTAMP WITH LOCAL TIME ZONE Data Type
- Inserting Values into Datetime Data Types
- Choosing a TIMESTAMP Data Type

DATE Data Type

The DATE data type stores date and time information. Although date and time information can be represented in both character and number data types, the DATE data type has special associated properties. For each DATE value, Oracle Database stores the following information: century, year, month, date, hour, minute, and second.

You can specify a date value by:

- Specifying the date value as a literal
- Converting a character or numeric value to a date value with the TO_DATE function

A date can be specified as an ANSI date literal or as an Oracle Database date value.

An ANSI date literal contains no time portion and must be specified in exactly the following format:

```
DATE 'YYYY-MM-DD'
```

The following is an example of an ANSI date literal:

```
DATE '1998-12-25'
```

Alternatively, you can specify an Oracle Database date value as shown in the following example:

```
TO_DATE('1998-DEC-25 17:30','YYYY-MON-DD HH24:MI','NLS_DATE_LANGUAGE=AMERICAN')
```

The default date format for an Oracle Database date value is derived from the NLS_DATE_FORMAT and NLS_DATE_LANGUAGE initialization parameters. The date format in the example includes a two-digit number for the day of the month, an abbreviation of the month name, the four digits of the year, and a 24-hour time designation. The specification for NLS_DATE_LANGUAGE is included because 'DEC' is not a valid value for MON in all locales.

Oracle Database automatically converts character values that are in the default date format into date values when they are used in date expressions.

If you specify a date value without a time component, then the default time is midnight. If you specify a date value without a date, then the default date is the first day of the current month.

Oracle Database DATE columns always contain fields for both date and time. If your queries use a date format without a time portion, then you must ensure that the time fields in the DATE column are set to midnight. You can use the TRUNC (date) SQL function to ensure that the time fields are set to midnight, or you can make the query a test of greater than or less than (<, <=, >=, or >) instead of equality or inequality (= or !=). Otherwise, Oracle Database may not return the query results you expect.

See Also:

- *Oracle Database SQL Language Reference* for more information about the DATE data type
- "NLS_DATE_FORMAT" on page 3-16
- "NLS_DATE_LANGUAGE" on page 3-17
- *Oracle Database SQL Language Reference* for more information about literals, format elements such as MM, and the TO_DATE function

TIMESTAMP Data Type

The TIMESTAMP data type is an extension of the DATE data type. It stores year, month, day, hour, minute, and second values. It also stores fractional seconds, which are not stored by the DATE data type.

Specify the TIMESTAMP data type as follows:

```
TIMESTAMP [(fractional_seconds_precision)]
```

fractional_seconds_precision is optional and specifies the number of digits in the fractional part of the SECOND datetime field. It can be a number in the range 0 to 9. The default is 6.

For example, '26-JUN-02 09:39:16.78' shows 16.78 seconds. The fractional seconds precision is 2 because there are 2 digits in '78'.

You can specify the TIMESTAMP literal in a format like the following:

```
TIMESTAMP 'YYYY-MM-DD HH24:MI:SS.FF'
```

Using the example format, specify TIMESTAMP as a literal as follows:

```
TIMESTAMP '1997-01-31 09:26:50.12'
```

The value of NLS_TIMESTAMP_FORMAT initialization parameter determines the timestamp format when a character string is converted to the TIMESTAMP data type. NLS_DATE_LANGUAGE determines the language used for character data such as MON.

See Also:

- *Oracle Database SQL Language Reference* for more information about the TIMESTAMP data type
- "NLS_TIMESTAMP_FORMAT" on page 3-18
- "NLS_DATE_LANGUAGE" on page 3-17

TIMESTAMP WITH TIME ZONE Data Type

TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a time zone region name or time zone offset in its value. The time zone offset is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time, formerly Greenwich Mean Time). Specify the TIMESTAMP WITH TIME ZONE data type as follows:

```
TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE
```

fractional_seconds_precision is optional and specifies the number of digits in the fractional part of the SECOND datetime field.

You can specify TIMESTAMP WITH TIME ZONE as a literal as follows:

```
TIMESTAMP '1997-01-31 09:26:56.66 +02:00'
```

Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of the TIME ZONE offsets stored in the data. For example, the following expressions have the same value:

```
TIMESTAMP '1999-01-15 8:00:00 -8:00'
TIMESTAMP '1999-01-15 11:00:00 -5:00'
```

You can replace the UTC offset with the TZR (time zone region) format element. The following expression specifies America/Los_Angeles for the time zone region:

```
TIMESTAMP '1999-01-15 8:00:00 America/Los_Angeles'
```

To eliminate the ambiguity of boundary cases when the time switches from Standard Time to Daylight Saving Time, use both the TZR format element and the corresponding TZD format element. The TZD format element is an abbreviation of the time zone region with Daylight Saving Time information included. Examples are PST for U. S. Pacific Standard Time and PDT for U. S. Pacific Daylight Time. The following specification ensures that a Daylight Saving Time value is returned:

```
TIMESTAMP '1999-10-29 01:30:00 America/Los_Angeles PDT'
```

If you do not add the TZD format element, and the datetime value is ambiguous, then Oracle Database returns an error if you have the ERROR_ON_OVERLAP_TIME session parameter set to TRUE. If ERROR_ON_OVERLAP_TIME is set to FALSE (the default value), then Oracle Database interprets the ambiguous datetime as Standard Time.

The default date format for the TIMESTAMP WITH TIME ZONE data type is determined by the value of the NLS_TIMESTAMP_TZ_FORMAT initialization parameter.

See Also:

- *Oracle Database SQL Language Reference* for more information about the TIMESTAMP WITH TIME ZONE data type
- "TIMESTAMP Data Type" on page 4-3 for more information about fractional seconds precision
- "Support for Daylight Saving Time" on page 4-30
- "NLS_TIMESTAMP_TZ_FORMAT" on page 3-19
- *Oracle Database SQL Language Reference* for more information about format elements
- *Oracle Database SQL Language Reference* for more information about setting the ERROR_ON_OVERLAP_TIME session parameter

TIMESTAMP WITH LOCAL TIME ZONE Data Type

TIMESTAMP WITH LOCAL TIME ZONE is another variant of TIMESTAMP. It differs from TIMESTAMP WITH TIME ZONE as follows: data stored in the database is normalized to the database time zone, and the time zone offset is not stored as part of the column data. When users retrieve the data, Oracle Database returns it in the users' local session time zone. The time zone offset is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time, formerly Greenwich Mean Time).

Specify the TIMESTAMP WITH LOCAL TIME ZONE data type as follows:

```
TIMESTAMP [(fractional_seconds_precision)] WITH LOCAL TIME ZONE
```

fractional_seconds_precision is optional and specifies the number of digits in the fractional part of the SECOND datetime field.

There is no literal for TIMESTAMP WITH LOCAL TIME ZONE, but TIMESTAMP literals and TIMESTAMP WITH TIME ZONE literals can be inserted into a TIMESTAMP WITH LOCAL TIME ZONE column.

The default date format for TIMESTAMP WITH LOCAL TIME ZONE is determined by the value of the NLS_TIMESTAMP_FORMAT initialization parameter.

See Also:

- *Oracle Database SQL Language Reference* for more information about the TIMESTAMP WITH LOCAL TIME ZONE data type
- "TIMESTAMP Data Type" on page 4-3 for more information about fractional seconds precision
- "NLS_TIMESTAMP_FORMAT" on page 3-18

Inserting Values into Datetime Data Types

You can insert values into a datetime column in the following ways:

- Insert a character string whose format is based on the appropriate NLS format value
- Insert a literal
- Insert a literal for which implicit conversion is performed
- Use the TO_TIMESTAMP, TO_TIMESTAMP_TZ, or TO_DATE SQL function

The following examples show how to insert data into datetime data types.

Example 4–1 Inserting Data into a DATE Column

Set the date format.

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-YYYY HH24:MI:SS';
```

Create a table table_dt with columns c_id and c_dt. The c_id column is of NUMBER data type and helps to identify the method by which the data is entered. The c_dt column is of DATE data type.

```
SQL> CREATE TABLE table_dt (c_id NUMBER, c_dt DATE);
```

Insert a date as a character string.

```
SQL> INSERT INTO table_dt VALUES(1, '01-JAN-2003');
```

Insert the same date as a DATE literal.

```
SQL> INSERT INTO table_dt VALUES(2, DATE '2003-01-01');
```

Insert the date as a TIMESTAMP literal. Oracle Database drops the time zone information.

```
SQL> INSERT INTO table_dt VALUES(3, TIMESTAMP '2003-01-01 00:00:00 America/Los_
Angeles');
```

Insert the date with the TO_DATE function.

```
SQL> INSERT INTO table_dt VALUES(4, TO_DATE('01-JAN-2003', 'DD-MON-YYYY'));
```

Display the data.

```
SQL> SELECT * FROM table_dt;

C_ID       C_DT
---------- --------------------
1          01-JAN-2003 00:00:00
2          01-JAN-2003 00:00:00
3          01-JAN-2003 00:00:00
4          01-JAN-2003 00:00:00
```

Example 4–2 Inserting Data into a TIMESTAMP Column

Set the timestamp format.

```
SQL> ALTER SESSION SET NLS_TIMESTAMP_FORMAT='DD-MON-YY HH:MI:SSXFF';
```

Create a table table_ts with columns c_id and c_ts. The c_id column is of NUMBER data type and helps to identify the method by which the data is entered. The c_ts column is of TIMESTAMP data type.

```
SQL> CREATE TABLE table_ts(c_id NUMBER, c_ts TIMESTAMP);
```

Insert a date and time as a character string.

```
SQL> INSERT INTO table_ts VALUES(1, '01-JAN-2003 2:00:00');
```

Insert the same date and time as a `TIMESTAMP` literal.

```
SQL> INSERT INTO table_ts VALUES(2, TIMESTAMP '2003-01-01 2:00:00');
```

Insert the same date and time as a `TIMESTAMP WITH TIME ZONE` literal. Oracle Database converts it to a `TIMESTAMP` value, which means that the time zone information is dropped.

```
SQL> INSERT INTO table_ts VALUES(3, TIMESTAMP '2003-01-01 2:00:00 -08:00');
```

Display the data.

```
SQL> SELECT * FROM table_ts;

C_ID       C_TS
----------  ----------------------------
1           01-JAN-03 02:00:00.000000 AM
2           01-JAN-03 02:00:00.000000 AM
3           01-JAN-03 02:00:00.000000 AM
```

Note that the three methods result in the same value being stored.

Example 4–3 Inserting Data into the TIMESTAMP WITH TIME ZONE Data Type

Set the timestamp format.

```
SQL> ALTER SESSION SET NLS_TIMESTAMP_TZ_FORMAT='DD-MON-RR HH:MI:SSXFF AM TZR';
```

Set the time zone to `'-07:00'`.

```
SQL> ALTER SESSION SET TIME_ZONE='-7:00';
```

Create a table `table_tstz` with columns `c_id` and `c_tstz`. The `c_id` column is of `NUMBER` data type and helps to identify the method by which the data is entered. The `c_tstz` column is of `TIMESTAMP WITH TIME ZONE` data type.

```
SQL> CREATE TABLE table_tstz (c_id NUMBER, c_tstz TIMESTAMP WITH TIME ZONE);
```

Insert a date and time as a character string.

```
SQL> INSERT INTO table_tstz VALUES(1, '01-JAN-2003 2:00:00 AM -07:00');
```

Insert the same date and time as a `TIMESTAMP` literal. Oracle Database converts it to a `TIMESTAMP WITH TIME ZONE` literal, which means that the session time zone is appended to the `TIMESTAMP` value.

```
SQL> INSERT INTO table_tstz VALUES(2, TIMESTAMP '2003-01-01 2:00:00');
```

Insert the same date and time as a `TIMESTAMP WITH TIME ZONE` literal.

```
SQL> INSERT INTO table_tstz VALUES(3, TIMESTAMP '2003-01-01 2:00:00 -8:00');
```

Display the data.

```
SQL> SELECT * FROM table_tstz;

C_ID       C_TSTZ
----------  ------------------------------------
```

```
1           01-JAN-03 02:00.00:000000 AM -07:00
2           01-JAN-03 02:00:00.000000 AM -07:00
3           01-JAN-03 02:00:00.000000 AM -08:00
```

Note that the time zone is different for method 3, because the time zone information was specified as part of the TIMESTAMP WITH TIME ZONE literal.

Example 4–4 Inserting Data into the TIMESTAMP WITH LOCAL TIME ZONE Data Type

Consider data that is being entered in Denver, Colorado, U.S.A., whose time zone is UTC-7.

```
SQL> ALTER SESSION SET TIME_ZONE='-07:00';
```

Create a table table_tsltz with columns c_id and c_tsltz. The c_id column is of NUMBER data type and helps to identify the method by which the data is entered. The c_tsltz column is of TIMESTAMP WITH LOCAL TIME ZONE data type.

```
SQL> CREATE TABLE table_tsltz (c_id NUMBER, c_tsltz TIMESTAMP WITH LOCAL TIME ZONE);
```

Insert a date and time as a character string.

```
SQL> INSERT INTO table_tsltz VALUES(1, '01-JAN-2003 2:00:00');
```

Insert the same data as a TIMESTAMP WITH LOCAL TIME ZONE literal.

```
SQL> INSERT INTO table_tsltz VALUES(2, TIMESTAMP '2003-01-01 2:00:00');
```

Insert the same data as a TIMESTAMP WITH TIME ZONE literal. Oracle Database converts the data to a TIMESTAMP WITH LOCAL TIME ZONE value. This means the time zone that is entered (-08:00) is converted to the session time zone value (-07:00).

```
SQL> INSERT INTO table_tsltz VALUES(3, TIMESTAMP '2003-01-01 2:00:00 -08:00');
```

Display the data.

```
SQL> SELECT * FROM table_tsltz;

C_ID        C_TSLTZ
----------  ---------------------------------
1           01-JAN-03 02.00.00.000000 AM
2           01-JAN-03 02.00.00.000000 AM
3           01-JAN-03 03.00.00.000000 AM
```

Note that the information that was entered as UTC-8 has been changed to the local time zone, changing the hour from 2 to 3.

> **See Also:** "Datetime SQL Functions" on page 4-12 for more information about the TO_TIMESTAMP or TO_TIMESTAMP_TZ SQL functions

Choosing a TIMESTAMP Data Type

Use the TIMESTAMP data type when you need a datetime value to record the time of an event without the time zone. For example, you can store information about the times when workers punch a time card in and out of their assembly line workstations. Because this is always a local time it is then not needed to store the timezone part. The TIMESTAMP data type uses 7 or 11 bytes of storage.

Use the TIMESTAMP WITH TIME ZONE data type when the datetime value represents a future local time or the time zone information must be recorded with the value. Consider a scheduled appointment in a local time. The future local time may need to

be adjusted if the time zone definition, such as daylight saving rule, changes. Otherwise, the value can become incorrect. This data type is most immune to such impact.

The TIMESTAMP WITH TIME ZONE data type requires 13 bytes of storage, or two more bytes of storage than the TIMESTAMP and TIMESTAMP WITH LOCAL TIME ZONE data types because it stores time zone information. The time zone is stored as a time zone region name or as an offset from UTC. The data is available for display or calculations without additional processing. A TIMESTAMP WITH TIME ZONE column cannot be used as a primary key. If an index is created on a TIMESTAMP WITH TIME ZONE column, it becomes a function-based index.

The TIMESTAMP WITH LOCAL TIME ZONE data type stores the timestamp without time zone information. It normalizes the data to the database time zone every time the data is sent to and from a client. It requires 11 bytes of storage.

The TIMESTAMP WITH LOCAL TIME ZONE data type is appropriate when the original time zone is of no interest, but the relative times of events are important and daylight saving adjustment does not have to be accurate. The time zone conversion that this data type performs to and from the database time zone is asymmetrical, due to the daylight saving adjustment. Because this data type does not preserve the time zone information, it does not distinguish values near the adjustment in fall, whether they are daylight saving time or standard time. This confusion between distinct instants can cause an application to behave unexpectedly, especially if the adjustment takes place during the normal working hours of a user.

Note that some regions, such as Brazil and Israel, that update their Daylight Saving Transition dates frequently and at irregular periods, are particularly susceptible to time zone adjustment issues. If time information from these regions is key to your application, you may want to consider using one of the other datetime types.

Interval Data Types

Interval data types store time durations. They are used primarily with analytic functions. For example, you can use them to calculate a moving average of stock prices. You must use interval data types to determine the values that correspond to a particular percentile. You can also use interval data types to update historical tables.

This section includes the following topics:

- INTERVAL YEAR TO MONTH Data Type
- INTERVAL DAY TO SECOND Data Type
- Inserting Values into Interval Data Types

> **See Also:** *Oracle Database Data Warehousing Guide* for more information about analytic functions, including moving averages and inverse percentiles

INTERVAL YEAR TO MONTH Data Type

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. Specify INTERVAL YEAR TO MONTH as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

year_precision is the number of digits in the YEAR datetime field. Accepted values are 0 to 9. The default value of *year_precision* is 2.

Interval values can be specified as literals. There are many ways to specify interval literals. The following is one example of specifying an interval of 123 years and 2 months. The year precision is 3.

```
INTERVAL '123-2' YEAR(3) TO MONTH
```

> **See Also:** *Oracle Database SQL Language Reference* for more information about specifying interval literals with the `INTERVAL YEAR TO MONTH` data type

INTERVAL DAY TO SECOND Data Type

`INTERVAL DAY TO SECOND` stores a period of time in terms of days, hours, minutes, and seconds. Specify this data type as follows:

```
INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds_precision)]
```

`day_precision` is the number of digits in the `DAY` datetime field. Accepted values are 0 to 9. The default is 2.

`fractional_seconds_precision` is the number of digits in the fractional part of the `SECOND` datetime field. Accepted values are 0 to 9. The default is 6.

The following is one example of specifying an interval of 4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second. The fractional second precision is 3.

```
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
```

Interval values can be specified as literals. There are many ways to specify interval literals.

> **See Also:** *Oracle Database SQL Language Reference* for more information about specifying interval literals with the `INTERVAL DAY TO SECOND` data type

Inserting Values into Interval Data Types

You can insert values into an interval column in the following ways:

- Insert an interval as a literal. For example:

  ```
  INSERT INTO table1 VALUES (INTERVAL '4-2' YEAR TO MONTH);
  ```

  This statement inserts an interval of 4 years and 2 months.

  Oracle Database recognizes literals for other ANSI interval types and converts the values to Oracle Database interval values.

- Use the `NUMTODSINTERVAL`, `NUMTOYMINTERVAL`, `TO_DSINTERVAL`, and `TO_YMINTERVAL` SQL functions.

  > **See Also:** "Datetime SQL Functions" on page 4-12

Datetime and Interval Arithmetic and Comparisons

This section includes the following topics:

Datetime and Interval Arithmetic

You can perform arithmetic operations on date (DATE), timestamp (TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE) and interval (INTERVAL DAY TO SECOND and INTERVAL YEAR TO MONTH) data. You can maintain the most precision in arithmetic operations by using a timestamp data type with an interval data type.

You can use NUMBER constants in arithmetic operations on date and timestamp values. Oracle Database internally converts timestamp values to date values before doing arithmetic operations on them with NUMBER constants. This means that information about fractional seconds is lost during operations that include both date and timestamp values. Oracle Database interprets NUMBER constants in datetime and interval expressions as number of days.

Each DATE value contains a time component. The result of many date operations includes a fraction. This fraction means a portion of one day. For example, 1.5 days is 36 hours. These fractions are also returned by Oracle Database built-in SQL functions for common operations on DATE data. For example, the built-in MONTHS_BETWEEN SQL function returns the number of months between two dates. The fractional portion of the result represents that portion of a 31-day month.

Oracle Database performs all timestamp arithmetic in UTC time. For TIMESTAMP WITH LOCAL TIME ZONE data, Oracle Database converts the datetime value from the database time zone to UTC and converts back to the database time zone after performing the arithmetic. For TIMESTAMP WITH TIME ZONE data, the datetime value is always in UTC, so no conversion is necessary.

See Also:

- *Oracle Database SQL Language Reference* for detailed information about datetime and interval arithmetic operations
- "Datetime SQL Functions" on page 4-12 for information about which functions cause implicit conversion to DATE

Datetime Comparisons

When you compare date and timestamp values, Oracle Database converts the data to the more precise data type before doing the comparison. For example, if you compare data of TIMESTAMP WITH TIME ZONE data type with data of TIMESTAMP data type, Oracle Database converts the TIMESTAMP data to TIMESTAMP WITH TIME ZONE, using the session time zone.

The order of precedence for converting date and timestamp data is as follows:

1. DATE
2. TIMESTAMP
3. TIMESTAMP WITH LOCAL TIME ZONE
4. TIMESTAMP WITH TIME ZONE

For any pair of data types, Oracle Database converts the data type that has a smaller number in the preceding list to the data type with the larger number.

Explicit Conversion of Datetime Data Types

If you want to do explicit conversion of datetime data types, use the CAST SQL function. You can explicitly convert DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE to another data type in the list.

See Also: *Oracle Database SQL Language Reference*

Datetime SQL Functions

Datetime functions operate on date (DATE), timestamp (TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE) and interval (INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH) values.

Some of the datetime functions were designed for the Oracle Database DATE data type. If you provide a timestamp value as their argument, then Oracle Database internally converts the input type to a DATE value. Oracle Database does not perform internal conversion for the ROUND and TRUNC functions.

Table 4–1 shows the datetime functions that were designed for the Oracle Database DATE data type. For more detailed descriptions, refer to *Oracle Database SQL Language Reference.*

Table 4–1 Datetime Functions Designed for the DATE Data Type

Function	Description
ADD_MONTHS	Returns the date *d* plus *n* months
LAST_DAY	Returns the last day of the month that contains *date*
MONTHS_BETWEEN	Returns the number of months between *date1* and *date2*
NEW_TIME	Returns the date and time in *zone2* time zone when the date and time in *zone1* time zone are *date* **Note:** This function takes as input only a limited number of time zones. You can have access to a much greater number of time zones by combining the FROM_TZ function and the datetime expression.
NEXT_DAY	Returns the date of the first weekday named by *char* that is later than *date*
ROUND(date)	Returns *date* rounded to the unit specified by the *fmt* format model
TRUNC(date)	Returns *date* with the time portion of the day truncated to the unit specified by the *fmt* format model

Table 4–2 describes additional datetime functions. For more detailed descriptions, refer to *Oracle Database SQL Language Reference.*

Table 4–2 Additional Datetime Functions

Datetime Function	Description
CURRENT_DATE	Returns the current date in the session time zone in a value in the Gregorian calendar, of the DATE data type
CURRENT_TIMESTAMP	Returns the current date and time in the session time zone as a TIMESTAMP WITH TIME ZONE value
DBTIMEZONE	Returns the value of the database time zone. The value is a time zone offset or a time zone region name
EXTRACT (datetime)	Extracts and returns the value of a specified datetime field from a datetime or interval value expression
FROM_TZ	Converts a TIMESTAMP value at a time zone to a TIMESTAMP WITH TIME ZONE value

Table 4–2 (Cont.) Additional Datetime Functions

Datetime Function	Description
LOCALTIMESTAMP	Returns the current date and time in the session time zone in a value of the TIMESTAMP data type
NUMTODSINTERVAL	Converts number *n* to an INTERVAL DAY TO SECOND literal
NUMTOYMINTERVAL	Converts number *n* to an INTERVAL YEAR TO MONTH literal
SESSIONTIMEZONE	Returns the value of the current session's time zone
SYS_EXTRACT_UTC	Extracts the UTC from a datetime with time zone offset
SYSDATE	Returns the date and time of the operating system on which the database resides, taking into account the time zone of the database server's operating system that was in effect when the database was started
SYSTIMESTAMP	Returns the system date, including fractional seconds and time zone of the system on which the database resides
TO_CHAR (datetime)	Converts a datetime or interval value of DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, or TIMESTAMP WITH LOCAL TIME ZONE data type to a value of VARCHAR2 data type in the format specified by the *fmt* date format
TO_DSINTERVAL	Converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of INTERVAL DAY TO SECOND data type
TO_NCHAR (datetime)	Converts a datetime or interval value of DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL MONTH TO YEAR, or INTERVAL DAY TO SECOND data type from the database character set to the national character set
TO_TIMESTAMP	Converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of TIMESTAMP data type
TO_TIMESTAMP_TZ	Converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the TIMESTAMP WITH TIME ZONE data type
TO_YMINTERVAL	Converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the INTERVAL YEAR TO MONTH data type
TZ_OFFSET	Returns the time zone offset that corresponds to the entered value, based on the date that the statement is executed

Table 4–3 describes functions that are used in the Daylight Saving Time (DST) upgrade process, and are only available when preparing or updating windows. For more detailed information, see *Oracle Database SQL Language Reference*.

Table 4–3 Time Zone Conversion Functions

Time Zone Function	Description
ORA_DST_AFFECTED	Enables you to verify whether the data in a column is affected by upgrading the DST rules from one version to another version
ORA_DST_CONVERT	Enables you to upgrade your TSTZ column data from one version to another
ORA_DST_ERROR	Enables you to verify that there are no errors when upgrading a datetime value

See Also: *Oracle Database SQL Language Reference*

Datetime and Time Zone Parameters and Environment Variables

This section includes the following topics:

- Datetime Format Parameters
- Time Zone Environment Variables
- Daylight Saving Time Session Parameter
- Daylight Saving Time Upgrade Parameter

Datetime Format Parameters

Table 4–4 contains the names and descriptions of the datetime format parameters.

Table 4–4 Datetime Format Parameters

Parameter	Description
NLS_DATE_FORMAT	Defines the default date format to use with the TO_CHAR and TO_DATE functions
NLS_TIMESTAMP_FORMAT	Defines the default timestamp format to use with the TO_CHAR and TO_TIMESTAMP functions
NLS_TIMESTAMP_TZ_FORMAT	Defines the default timestamp with time zone format to use with the TO_CHAR and TO_TIMESTAMP_TZ functions

Their default values are derived from NLS_TERRITORY.

You can specify their values by setting them in the initialization parameter file. If you change the values in the initialization parameter file, you must restart the instance for the change to take effect. You can also specify their values for a client as client environment variables. For Java clients, the value of NLS_TERRITORY is derived from the default locale of JRE. The values specified in the initialization parameter file are not used for JDBC sessions.

To change their values during a session, use the ALTER SESSION statement.

See Also:

- "Date and Time Parameters" on page 3-15 for more information, including examples
- "NLS_DATE_FORMAT" on page 3-16
- "NLS_TIMESTAMP_FORMAT" on page 3-18
- "NLS_TIMESTAMP_TZ_FORMAT" on page 3-19

Time Zone Environment Variables

The time zone environment variables are:

- ORA_TZFILE, which enables you to specify a time zone on the client and Oracle Database server. Note that when you specify ORA_TZFILE on Oracle Database server, the only time when this environment variable takes effect is during the creation of the database.
- ORA_SDTZ, which specifies the default session time zone.

See Also:

- "Choosing a Time Zone File" on page 4-15
- "Setting the Session Time Zone" on page 4-28

Daylight Saving Time Session Parameter

`ERROR_ON_OVERLAP_TIME` is a session parameter that determines how Oracle Database handles an ambiguous datetime boundary value. Ambiguous datetime values can occur when the time changes between Daylight Saving Time and standard time.

The possible values are `TRUE` and `FALSE`. When `ERROR_ON_OVERLAP_TIME` is `TRUE`, then an error is returned when Oracle Database encounters an ambiguous datetime value. When `ERROR_ON_OVERLAP_TIME` is `FALSE`, then ambiguous datetime values are assumed to be the standard time representation for the region. The default value is `FALSE`.

See Also:

- "Support for Daylight Saving Time" on page 4-30
- *Oracle Database SQL Language Reference*

Daylight Saving Time Upgrade Parameter

`DST_UPGRADE_INSERT_CONV` is an initialization parameter that is only used during the upgrade window of the Daylight Saving Time (DST) upgrade process. It is only applicable to tables with `TIMESTAMP WITH TIME ZONE` columns because those are the only ones that are modified during the DST upgrade.

During the upgrade window of the DST patching process (which is described in the `DBMS_DST` package), tables with `TIMESTAMP WITH TIMEZONE` data undergo conversion to the new time zone version. Columns in tables that have not yet been converted will still have the `TIMESTAMP WITH TIMEZONE` reflecting the previous time zone version. In order to present the data in these columns as though they had been converted to the new time zone version when you issue `SELECT` statements, Oracle adds by default conversion operators over the columns to convert them to the new version. Adding the conversion operator may, however, slow down queries and disable usage of indexes on the `TIMESTAMP WITH TIMEZONE` columns. Hence, Oracle provides a parameter, `DST_UPGRADE_INSERT_CONV`, that specifies whether or not internal operators are allocated on top of `TIMESTAMP WITH TIMEZONE` columns of tables that have not been upgraded. By default, its value is `TRUE`. If users know that the conversion process will not affect the `TIMESTAMP WITH TIMEZONE` columns, then this parameter can be set to `FALSE`.

Oracle strongly recommends that you set this parameter to `TRUE` throughout the DST patching process. By default, this parameter is set to `TRUE`. However, if set to `TRUE`, query performance may be degraded on unconverted tables. Note that this only applies during the upgrade window.

See Also:

- *Oracle Database Reference*
- *Oracle Database PL/SQL Packages and Types Reference*

Choosing a Time Zone File

The Oracle Database time zone files contain the valid time zone names. The following information is also included for each time zone:

- Offset from Coordinated Universal Time (UTC)
- Transition times for Daylight Saving Time
- Abbreviations for standard time and Daylight Saving Time

Oracle Database supplies multiple versions of time zone files, and there are two types of file associated with each one: a large file, which contains all the time zones defined in the database, and a small file, which contains only the most commonly used time zones. The large versions are designated as `timezlrg_<version_number>.dat`, while the small versions are designated as `timezone_<version_number>.dat`. The files are located in the `oracore/zoneinfo` subdirectory under the Oracle Database home directory, so, for example, the default time zone file is the highest version time zone file in this subdirectory. For example, in Oracle Database 11*g*, release 2, the default file is `$ORACLE_HOME/oracore/zoneinfo/timezlrg_14.dat`, which contains all the time zones defined in the database.

Examples of time zone files are:

```
$ORACLE_HOME/oracore/zoneinfo/timezlrg_4.dat   -- large version 4
$ORACLE_HOME/oracore/zoneinfo/timezone_4.dat   -- small version 4
$ORACLE_HOME/oracore/zoneinfo/timezlrg_5.dat   -- large version 5
$ORACLE_HOME/oracore/zoneinfo/timezone_5.dat   -- small version 5
```

During the database creation process, you choose the time zone version for the server. This version is fixed, but you can, however, go through the upgrade process to achieve a higher version. You can use different versions of time zone files on the client and server, but Oracle recommends that you do not. This is because there is a performance penalty when a client on one version communicates with a server on a different version. The performance penalty arises because the `TIMESTAMP WITH TIME ZONE` (TSTZ) data is transferred using a local timestamp instead of UTC.

On the server, Oracle Database always uses a large file. On the client, you can use either a large or a small file. If you use a large time zone file on the client, then you should continue to use it unless you are sure that no information will be missing if you switch to a smaller one. If you use a small file, you have to make sure that the client does not query data that is not present in the small time zone file. Otherwise, you get an error.

You can enable the use of a specific time zone file in the client or on the server. If you want to enable a time zone file on the server, there are two situations. One is that you go through a time zone upgrade to the target version. See "Upgrading the Time Zone File and Timestamp with Time Zone Data" on page 4-18 for more information. Another is when you are creating a new database, in that case, you can set the `ORA_TZFILE` environment variable to point to the time zone file of your choice.

To enable a specific time zone file on the client, you can set `ORA_TZFILE` to whatever time zone file you want. If `ORA_TZFILE` is not set, Oracle Database automatically picks up and use the file with the latest time zone version. See *Oracle Call Interface Programmer's Guide* for more information on how to upgrade Daylight Saving Time on the client.

Oracle Database time zone data is derived from the public domain information available at `http://www.iana.org/time-zones`. Oracle Database time zone data may not reflect the most recent data available at this site.

You can obtain a list of time zone names and time zone abbreviations from the time zone file that is installed with your database by entering the following statement:

```
SELECT TZNAME, TZABBREV
FROM V$TIMEZONE_NAMES
```

```
ORDER BY TZNAME, TZABBREV;
```

For the default time zone file, this statement results in output similar to the following:

```
TZNAME               TZABBREV
-------------------- ----------
Africa/Abidjan       GMT
Africa/Abidjan       LMT
...
Africa/Algiers       CEST
Africa/Algiers       CET
Africa/Algiers       LMT
Africa/Algiers       PMT
Africa/Algiers       WET
Africa/Algiers       WEST
...
WET                  LMT
WET                  WEST
WET                  WET

2137 rows selected.
```

In the above output, 2 time zone abbreviations are associated with the Africa/Abidjan time zone, and 6 abbreviations are associated with the Africa/Algiers time zone. The following table shows some of the time zone abbreviations and their meanings.

Time Zone Abbreviation	Meaning
LMT	Local Mean Time
PMT	Paris Mean Time
WET	Western European Time
WEST	Western European Summer Time
CET	Central Europe Time
CEST	Central Europe Summer Time
EET	Eastern Europe Time
EEST	Eastern Europe Summer Time

Note that an abbreviation can be associated with multiple time zones. For example, CET is associated with both Africa/Algiers and Africa/Casablanca, as well as time zones in Europe.

If you want a list of time zones without repeating the time zone name for each abbreviation, use the following query:

```
SELECT UNIQUE TZNAME
FROM V$TIMEZONE_NAMES;
```

For example, version 11 contains output similar to the following:

```
TZNAME
--------------------
Africa/Addis_Ababa
Africa/Bissau
Africa/Ceuta
...
Turkey
US/East-Indiana
```

```
US/Samoa
```

The default time zone file contains more than 350 unique time zone names. The small time zone file contains more than 180 unique time zone names.

See Also:

- "Time Zone Region Names" on page A-25 for a list of valid Oracle Database time zone names
- `$ORACLE_HOME/oracore/zoneinfo/timezdif.csv`, provided with your Oracle Database software installation, for a full list of time zones changed since Oracle9*i*
- *Oracle Database Upgrade Guide* for upgrade information

Upgrading the Time Zone File and Timestamp with Time Zone Data

The time zone files that are supplied with the Oracle Database are updated periodically to reflect changes in transition rules for various time zone regions. To find which Time Zone File your database currently uses, query `V$TIMEZONE_FILE`.

Note: Oracle Database 9*i* includes version 1 of the time zone files, and Oracle Database 10*g* includes version 2. For Oracle Database 11*g*, release 2, all time zone files from versions 1 to 14 are included. Various patches and patch sets, which are released separately for these releases, may update the time zone file version as well. With Oracle Database 12*c*, version 18 is included.

Daylight Saving Time (DST) Transition Rules Changes

Governments can and do change the rules for when Daylight Saving Time takes effect or how it is handled. When this occurs, Oracle provides a new set of transition rules for handling timestamp with time zone data.

Transition periods for the beginning or ending of Daylight Saving Time can potentially introduce problems (such as data loss) when handling timestamps with time zone data. Because of this, there are certain rules for dealing with the transition, and, in this release, these transition rules have changed. In addition, Oracle has significantly improved the way of dealing with this transition by providing a new package called `DBMS_DST`.

The changes to DST transition rules may affect existing data of `TIMESTAMP WITH TIME ZONE` data type, because of the way Oracle Database stores this data internally. When users enter timestamps with time zone, Oracle Database converts the data to UTC, based on the transition rules in the time zone file, and stores the data together with the ID of the original time zone on disk. When data is retrieved, the reverse conversion from UTC takes place. For example, when the version 2 transition rules were in effect, the value `TIMESTAMP '2007-11-02 12:00:00 America/Los_Angeles'`, would have been stored as UTC value `'2007-11-02 20:00:00'` *plus* the time zone ID for `'America/Los_Angeles'`. The time in Los Angeles would have been UTC *minus* eight hours (PST). Under version 3 of the transition rules, the offset for the same day is *minus* seven hours (PDT). Beginning with year 2007, the DST has been in effect longer (until the first Sunday of November, which is November 4th in 2007). Now, when users retrieve the same timestamp and the new offset is added to the stored UTC time, they receive `TIMESTAMP '2007-11-02 13:00:00 America/Los_Angeles'`. There is a one hour difference compared to the data previous to version 3 taking effect.

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the DBMS_DST package.

> **Note:** For any time zone region whose transition rules have been updated, the upgrade process discussed throughout this section, "Upgrading the Time Zone File and Timestamp with Time Zone Data" on page 4-18, affects only timestamps that point to the future relative to the effective date of the corresponding DST rule change. For example, no timestamp before year 2007 is affected by the version 3 change to the 'America/Los_Angeles' time zone region.

Preparing to Upgrade the Time Zone File and Timestamp with Time Zone Data

Before you actually upgrade any data, you should verify what the impact of the upgrade is likely to be. In general, you can consider the upgrade process to have two separate subprocesses. The first is to create a prepare window and the second is to create an upgrade window. The prepare window is the time where you check how much data has to be updated in the database, while the upgrade window is the time when the upgrade actually occurs.

While not required, Oracle strongly recommends you perform the prepare window step. In addition to finding out how much data will have to be modified during the upgrade, thus giving you an estimate of how much time the upgrade will take, you will also see any semantic errors that you may encounter. See "Error Handling when Upgrading Time Zone File and Timestamp with Time Zone Data" on page 4-26.

You can create a prepare window to find the affected data using the following steps:

1. Install the desired (latest) time zone file to which you will be later migrating into $ORACLE_HOME/oracore/zoneinfo. The desired (latest) version of timezlrg_<version_number>.dat is required, while timezone_<version_number>.dat may also be added at your discretion. These files can be found on My Oracle Support.
2. You can optionally create an error table as well as a table that contains affected timestamp with time zone information by using DBMS_DST.CREATE_ERROR_TABLE and DBMS_DST.CREATE_AFFECTED_TABLE, respectively, If you do not, Oracle Database uses the pre-built sys.dst$affected_tables and sys.dst$error_table. These tables are used in step 4.

   ```
   EXEC DBMS_DST.CREATE_AFFECTED_TABLE('my_affected_tables');
   EXEC DBMS_DST.CREATE_ERROR_TABLE('my_error_table');
   ```

3. Invoke DBMS_DST.BEGIN_PREPARE(<new_version>), which is the version you chose in Step 1. See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding DBMS_DST privilege information.
4. Check the affected data by invoking DBMS_DST.FIND_AFFECTED_TABLES, and verifying the affected columns by querying sys.dst$affected_tables. Also, it is particularly important to check sys.dst$affected_tables.error_count for possible errors. If the error count is greater than 0, you can check what kind of errors might expect during the upgrade by checking sys.dst$error_table. See "Error Handling when Upgrading Time Zone File and Timestamp with Time Zone Data" on page 4-26.
5. End the prepare window by invoking DBMS_DST.END_PREPARE.

Note: Note that only one DBA should run the prepare window at one time. Also, make sure to correct all errors before running the upgrade.

Note: You can find the matrix of available patches for updating your time zone files by going to Oracle Support and reading Document ID 412160.1.

Steps to Upgrade Time Zone File and Timestamp with Time Zone Data

Upgrading a time zone file and timestamp with time zone data contains the following steps:

1. If you have not already done so, download the desired (latest) version of `timezlrg_<version_number>.dat` and install it in `$ORACLE_HOME/oracore/zoneinfo`. In addition, you can optionally download `timezone_<version_number>.dat` from My Oracle Support and put it in the same location.
2. Shut down the database. In Oracle RAC, you must shut down all instances.
3. Start up the database in `UPGRADE` mode. Note that, in Oracle RAC, only one instance should be started. See *Oracle Database Upgrade Guide* for an explanation of `UPGRADE` mode.
4. Execute `DBMS_DST.BEGIN_UPGRADE(<new_version>)`. Optionally, you can have two other parameters that you can specify to `TRUE` if you do not want to ignore semantic errors during the upgrade of dictionary tables that contain timestamp with time zone data. If you specify `TRUE` for either or both of these parameters, the errors are populated into `sys.dst$error_table` by default. In this case, you might want to truncate the error table before you begin the `BEGIN_UPGRADE` procedure. See *Oracle Database PL/SQL Packages and Types Reference* for more information.
5. If the `BEGIN_UPGRADE` execution fails, an ORA-56927 error (Starting an upgrade window failed) is raised.

   After `BEGIN_UPGRADE` finishes executing with errors, check `sys.dst$error_table` to determine if the dictionary conversion was successful. If successful, there will not be any rows in the table. If there are errors, correct these errors manually and rerun `DBMS_DST.BEGIN_UPGRADE(<new_version>)`. See "Error Handling when Upgrading Time Zone File and Timestamp with Time Zone Data" on page 4-26.
6. Restart the database in normal mode.
7. Truncate the error and trigger tables (by default, `sys.dst$error_table` and `sys.dst$trigger_table`).

   The trigger table records the disabled TSTZ table triggers during the upgrade process, which is passed as a parameter to `DBMS_DST.UPGRADE_*` procedures. Note that you can optionally create your own trigger table by calling `DBMS_DST.CREATE_TRIGGER_TABLE`. During `DBMS_DST.UPGRADE_*`, Oracle Database first disables the triggers on a TSTZ table before performing the upgrade of its affected TSTZ data. Oracle Database also saves the information from those triggers in `sys.dst$trigger_table`. After completing the upgrade of the affected TSTZ data in the table, those disabled triggers are enabled by reading from `sys.dst$trigger_table` and then removed from `sys.dst$trigger_table`. If any fatal error occurs, such as an unexpected instance shutdown during the upgrade

process, you should check `sys.dst$trigger_table` to see if any trigger has not been restored to its previous active state before the upgrade.

8. Upgrade the TSTZ data in all tables by invoking `DBMS_DST.UPGRADE_DATABASE`.
9. Verify that all tables have finished being upgraded by querying the `DBA_TSTZ_TABLES` view, as shown in "Example of Updating Daylight Saving Time Behavior" on page 4-21. Then look at `dst$error_table` to see if there were any errors. If there were errors, correct the errors and rerun `DBMS_DST.UPGRADE_TABLE` on the relevant tables. Or, if you do not think those errors are important, re-run with the parameters set to ignore errors.
10. End the upgrade window by invoking `DBMS_DST.END_UPGRADE`.

Note: Tables containing timestamp with time zone columns need to be in a state where they can be updated. So, as an example, the columns cannot have validated and disabled check constraints as this prevents updating.

Oracle recommends that you use the parallel option if a table size is greater than 2 Gigabytes. Oracle also recommends that you allow Oracle to handle any semantic errors that may arise.

Note that, when you issue a `CREATE` statement for error, trigger, or affected tables, you must pass the table name only, not the schema name. This is because the tables will be created in the current invoker's schema.

Example of Updating Daylight Saving Time Behavior

This example illustrates updating DST behavior to Oracle Database 11*g*, release 2. First, assume that your current database is using time zone version 3, and also assume you have an existing table `t`, which contains timestamp with time zone data.

```
CONNECT scott/tiger
DROP TABLE t;
CREATE TABLE t (c NUMBER, mark VARCHAR(25), ts TIMESTAMP WITH TIME ZONE);

INSERT INTO t VALUES(1, 'not_affected',
                    to_timestamp_tz('22-sep-2006 13:00:00 america/los_angeles',
                                    'dd-mon-yyyy hh24:mi:ss tzr tzd'));
INSERT INTO t VALUES(4, 'affected_err_exist',
                     to_timestamp_tz('11-mar-2007 00:30:00 america/st_johns',
                                    'dd-mon-yyyy hh24:mi:ss tzr tzd'));
INSERT INTO t VALUES(6, 'affected_no_err',
                     to_timestamp_tz('11-mar-2007 01:30:00 america/st_johns',
                                    'dd-mon-yyyy hh24:mi:ss tzr tzd'));
INSERT INTO t VALUES(14, 'affected_err_dup',
                     to_timestamp_tz('21-sep-2006 23:30:00 egypt',
                                    'dd-mon-yyyy hh24:mi:ss tzr tzd'));
COMMIT;
```

Then, optionally, you can start a prepare window to check the affected data and potential semantic errors where there is an overlap or non-existing time. To do this, you should start a window for preparation to migrate to version 4. It is assumed that you have the necessary privileges. These privileges are controlled with the `DBMS_DST` package. See *Oracle Database PL/SQL Packages and Types Reference* for more information.

As an example, first, prepare the window.

```
conn / AS SYSDBA
set serveroutput on
EXEC DBMS_DST.BEGIN_PREPARE(4);

A prepare window has been successfully started.

PL/SQL procedure successfully completed.
```

Note that the argument 4 causes version 4 to be used in this statement. After this window is successfully started, you can check the status of the DST in DATABASE_PROPERTIES, as in the following:

```
SELECT PROPERTY_NAME, SUBSTR(property_value, 1, 30) value
FROM DATABASE_PROPERTIES
WHERE NAME LIKE 'DST_%'
ORDER BY PROPERTY_NAME;
```

You will see output resembling the following:

```
PROPERTY_NAME                  VALUE
-----------------------------  ---------
DST_PRIMARY_TT_VERSION         3
DST_SECONDARY_TT_VERSION       4
DST_UPGRADE_STATE              PREPARE
```

Next, you can invoke DBMS_DST.FIND_AFFECTED_TABLES to find all the tables in the database that are affected if you upgrade from version 3 to version 4. This table contains the table owner, table name, column name, row count, and error count. Here, you have the choice of using the defaults for error tables (sys.dst$error_table) and affected tables (sys.dst$affected_table) or you can create your own. In this example, we create our own tables by using DBMS_DST.CREATE_ERROR_TABLE and DBMS_DST.CREATE_AFFECTED_TABLE and then pass to FIND_AFFECTED_TABLES, as in the following:

```
EXEC DBMS_DST.CREATE_AFFECTED_TABLE('scott.my_affected_tables');
EXEC DBMS_DST.CREATE_ERROR_TABLE('scott.my_error_table');
```

It is a good idea to make sure that there are no rows in these tables. You can do this by truncating the tables, as in the following:

```
TRUNCATE TABLE scott.my_affected_tables;
TRUNCATE TABLE scott.my_error_table;
```

Then, you can invoke FIND_AFFECTED_TABLES to see which tables are impacted during the upgrade:

```
EXEC DBMS_DST.FIND_AFFECTED_TABLES(affected_tables  => 'scott.my_affected_tables',
                                   log_errors       => TRUE,
                                   log_errors_table => 'scott.my_error_table');
```

Then, check the affected tables:

```
SELECT * FROM scott.my_affected_tables;

TABLE_OWNER   TABLE_NAME   COLUMN_NAM   ROW_COUNT  ERROR_COUNT
-----------   ----------   ----------   ---------  -----------
SCOTT         T            TS                   3            2
```

Then, check the error table:

```
SELECT * FROM scott.my_error_table;
```

```
TABLE_OWNER   TABLE_NAME  COLUMN_NAME   ROWID                ERROR_NUMBER
-----------   ----------  -----------   ------------------   ------------
SCOTT         T           TS            AAAPW3AABAAANzoAAB           1878
SCOTT         T           TS            AAAPW3AABAAANzoAAE           1883
```

These errors can be corrected by seeing "Error Handling when Upgrading Time Zone File and Timestamp with Time Zone Data" on page 4-26. Then, end the prepare window, as in the following statement:

```
EXEC DBMS_DST.END_PREPARE;

A prepare window has been successfully ended.

PL/SQL procedure successfully completed.
```

After this, you can check the DST status in DATABASE_PROPERTIES:

```
SELECT PROPERTY_NAME, SUBSTR(property_value, 1, 30) value
FROM DATABASE_PROPERTIES
WHERE PROPERTY_NAME LIKE 'DST_%'
ORDER BY PROPERTY_NAME;

PROPERTY_NAME                 VALUE
-------------------------     --------
DST_PRIMARY_TT_VERSION        3
DST_SECONDARY_TT_VERSION      0
DST_UPGRADE_STATE             NONE
```

Next, you can use the upgrade window to upgrade the affected data. To do this, first, start an upgrade window. Note that the database must be opened in UPGRADE mode before you can execute DBMS_DST.BEGIN_UPGRADE. In Oracle RAC, only one instance can be started. BEGIN_UPGRADE upgrades all dictionary tables in one transaction, so the invocation will either succeed or fail as one whole. During the procedure's execution, all user tables with TSTZ data are marked as an upgrade in progress. See *Oracle Database Upgrade Guide* for more information.

Also, only SYSDBA can start an upgrade window. If you do not open the database in UPGRADE mode and invoke BEGIN_UPGRADE, you will see the following error:

```
EXEC DBMS_DST.BEGIN_UPGRADE(4);
BEGIN DBMS_DST.BEGIN_UPGRADE(4); END;

*
ERROR at line 1:
ORA-56926: database must be in UPGRADE mode in order to start an upgrade window
ORA-06512: at "SYS.DBMS_SYS_ERROR", line 79
ORA-06512: at "SYS.DBMS_DST", line 1021
ORA-06512: at line 1
```

So, BEGIN_UPGRADE upgrades system tables that contain TSTZ data and marks user tables (containing TSTZ data) with the UPGRADE_IN_PROGRESS property. This can be checked in ALL_TSTZ_TABLES, and is illustrated later in this example.

There are two parameters in BEGIN_UPGRADE that are for handling semantic errors: error_on_overlap_time (error number ORA-1883) and error_on_nonexisting_time (error number ORA-1878). If the parameters use the default setting of FALSE, Oracle converts the data using a default conversion and does not signal an error. See "Error Handling when Upgrading Time Zone File and Timestamp with Time Zone Data" on page 4-26 for more information regarding what they mean, and how to handle errors.

The following call can automatically correct semantic errors based on some default values when you upgrade the dictionary tables. If you do not ignore semantic errors, and you do have such errors in the dictionary tables, BEGIN_UPGRADE will fail. These semantic errors are populated into sys.dst$error_table.

```
EXEC DBMS_DST.BEGIN_UPGRADE(4);
An upgrade window has been successfully started.

PL/SQL procedure successfully completed.
```

After this, you can check the DST status in DATABASE_PROPERTIES, as in the following:

```
SELECT PROPERTY_NAME, SUBSTR(property_value, 1, 30) value
FROM DATABASE_PROPERTIES
WHERE PROPERTY_NAME LIKE 'DST_%'
ORDER BY PROPERTY_NAME;

PROPERTY_NAME                VALUE
-------------------------    --------
DST_PRIMARY_TT_VERSION              4
DST_SECONDARY_TT_VERSION            3
DST_UPGRADE_STATE             UPGRADE
```

Then, check which user tables are marked with UPGRADE_IN_PROGRESS:

```
SELECT OWNER, TABLE_NAME, UPGRADE_IN_PROGRESS FROM ALL_TSTZ_TABLES;

OWNER   TABLE_NAME                     UPGRADE_IN_PROGRESS
-----   -------------------------      -------------------
SYS     WRI$_OPTSTAT_AUX_HISTORY       NO
SYS     WRI$_OPTSTAT_OPR               NO
SYS     OPTSTAT_HIST_CONTROL$          NO
SYS     SCHEDULER$_JOB                 NO
SYS     KET$_AUTOTASK_STATUS           NO
SYS     AQ$_ALERT_QT_S                 NO
SYS     AQ$_KUPC$DATAPUMP_QUETAB_S     NO
DBSNMP  MGMT_DB_FEATURE_LOG            NO
WMSYS   WM$VERSIONED_TABLES            NO
SYS     WRI$_OPTSTAT_IND_HISTORY       NO
SYS     OPTSTAT_USER_PREFS$            NO
SYS     FGR$_FILE_GROUP_FILES          NO
SYS     SCHEDULER$_WINDOW              NO
SYS     WRR$_REPLAY_DIVERGENCE         NO
SCOTT   T                              YES
IX      AQ$_ORDERS_QUEUETABLE_S        YES
...
```

In this output, dictionary tables (in the SYS schema) have already been upgraded by BEGIN_UPGRADE. User tables, such as SCOTT.T, have not been and are in progress.

Now you can perform an upgrade of user tables using DBMS_DST.UPGRADE_DATABASE. All tables must be upgraded, otherwise, you will not be able to end the upgrade window using the END_UPGRADE procedure. Before this step, you must restart the database in normal mode. An example of the syntax is as follows:

```
VAR numfail number
BEGIN
   DBMS_DST.UPGRADE_DATABASE(:numfail,
            parallel                 => TRUE,
            log_errors               => TRUE,
            log_errors_table         => 'SYS.DST$ERROR_TABLE',
```

```
           log_triggers_table        => 'SYS.DST$TRIGGER_TABLE',
           error_on_overlap_time     => TRUE,
           error_on_nonexisting_time => TRUE);
DBMS_OUTPUT.PUT_LINE('Failures:'|| :numfail);
END;
/
```

If there are any errors, you should correct them and use `UPGRADE_TABLE` on the individual tables. In that case, you may need to handle tables related to materialized views, such as materialized view base tables, materialized view log tables, and materialized view container tables. There are a couple of considerations to keep in mind when upgrading these tables. First, the base table and its materialized view log table have to be upgraded atomically. Next, the materialized view container table has to be upgraded after all its base tables and the materialized view log tables have been upgraded. In general, Oracle recommends that you handle semantic errors by letting Oracle Database take the default action.

For the sake of this example, let us assume there were some errors in `SCOTT.T` after you ran `UPGRADE_DATABASE`. In that case, you can check these errors by issuing:

```
SELECT * FROM scott.error_table;

TABLE_OWNER   TABLE_NAME  COLUMN_NAME   ROWID                ERROR_NUMBER
-----------   ----------  -----------   ------------------   ------------
SCOTT         T           TS            AAAO2XAABAAANrgAAD           1878
SCOTT         T           TS            AAAO2XAABAAANrgAAE           1878
```

From this output, you can see that error number 1878 has occurred. This error means that an error has been thrown for a non-existing time.

To continue with this example, assume that `SCOTT.T` has a materialized view log `scott.mlog$_t`, and that there is a single materialized view on `SCOTT.T`. Then, assume that this 1878 error has been corrected.

Finally, you can upgrade the table, materialized view log and materialized view as follows:

```
BEGIN
  DBMS_DST.UPGRADE_TABLE(:numfail,
           table_list                => 'SCOTT.t, SCOTT.mlog$_T',
           parallel                  => TRUE,
           continue_after_errors     => FALSE,
           log_errors                => TRUE,
           log_errors_table          => 'SYS.DST$ERROR_TABLE',
           error_on_overlap_time     => FALSE,
           error_on_nonexisting_time => TRUE,
           log_triggers_table        => 'SYS.DST$TRIGGER_TABLE',
           atomic_upgrade            => TRUE);

 DBMS_OUTPUT.PUT_LINE('Failures:'|| :numfail);
END;
/

BEGIN
  DBMS_DST.UPGRADE_TABLE(:numfail,
           table_list                => 'SCOTT.MYMV_T',
           parallel                  => TRUE,
           continue_after_errors     => FALSE,
           log_errors                => TRUE,
           log_errors_table          => 'SYS.DST$ERROR_TABLE',
           error_on_overlap_time     => FALSE,
```

```
          error_on_nonexisting_time => TRUE,
          log_triggers_table        => 'SYS.DST$TRIGGER_TABLE',
          atomic_upgrade            => TRUE);

  DBMS_OUTPUT.PUT_LINE('Failures:'|| :numfail);
END;
/
```

The `atomic_upgrade` parameter enables you to combine the upgrade of the table with its materialized view log. See *Oracle Database PL/SQL Packages and Types Reference* for more information.

After all the tables are upgraded, you can invoke `END_UPGRADE` to end an upgrade window, as in the following:

```
BEGIN
  DBMS_DST.END_UPGRADE(:numfail);
END;
/
```

If no other table was upgraded successfully, the `END_UPGRADE` statement fails.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the `DBMS_DST` package

Error Handling when Upgrading Time Zone File and Timestamp with Time Zone Data

There are three major semantic errors that can occur during an upgrade. The first is when an existing time becomes a non-existing time, the second is when a time becomes duplicated, and the third is when a duplicate time becomes a non-duplicate time.

As an example of the first case, consider the change from Pacific Standard Time (PST) to Pacific Daylight Saving Time (PDT) in 2007. This change takes place on Mar-11-2007 at 2AM according to version 4 when the clock moves forward one hour to 3AM and produces a gap between 2AM and 3AM. In version 2, this time change occurs on Apr-01-2007. If you upgrade the time zone file from version 2 to version 4, any time in the interval between 2AM and 3AM on Mar-11-2007 is not valid, and raises an error (ORA-1878) if `ERROR_ON_NONEXISTING_TIME` is set to `TRUE`. Therefore, there is a non-existing time problem. When `ERROR_ON_NONEXISTING_TIME` is set to `FALSE`, which is the default value for this parameter, the error is not reported and Oracle preserves UTC time in this case. For example, "Mar-11-2007 02:30 PST" in version 2 becomes "Mar-11-2007 03:30 PDT" in version 4 as they both are translated to the same UTC timestamp.

An example of the second case occurs when changing from PDT to PST. For example, in version 4 for 2007, the change occurs on Nov-04-2007, when the time falls back from 2AM to 1AM. This means that times in the interval <1AM, 2AM> on Nov-04-2007 can appear twice, once with PST and once with PDT. In version 2, this transition occurs on Oct-28-2007 at 2AM. Thus, any timestamp within <1AM, 2AM> on Nov-04-2007, which is uniquely identified in version 2, results in an error (ORA-1883) in version 4, if `ERROR_ON_OVERLAP_TIME` is set to `TRUE`. If you leave this parameter on its default setting of `FALSE`, then the UTC timestamp value is preserved and no error is raised. In this situation, if you change the version from 2 to 4, timestamp "Nov-04-2007 01:30 PST" in version 2 becomes "Nov-04-2007 01:30 PST" in version 4.

The third case happens when a duplicate time becomes a non-duplicate time. Consider the transition from PDT to PST in 2007, for example, where <1AM, 2AM> on Oct-28-2007 in version 2 is the overlapped interval. Then both "Oct-28-2007 01:30 PDT"

and "Oct-28-2007 01:30 PST" are valid timestamps in version 2. If ERROR_ON_OVERLAP_TIME is set to TRUE, an ORA-1883 error is raised during an upgrade from version 2 to version 4. If ERROR_ON_OVERLAP_TIME is set to FALSE (the default value of this parameter), however, the LOCAL time is preserved and no error is reported. In this case, both "Oct-28-2007 01:30 PDT" and "Oct-28-2007 01:30 PST" in version 2 convert to the same "Oct-28-2007 01:30 PDT" in version 4. Note that setting ERROR_ON_OVERLAP_TIME to FALSE can potentially cause some time sequences to be reversed. For example, a job (Job A) scheduled at "Oct-28-2007 01:45 PDT" in version 2 is supposed to be executed before a job (Job B) scheduled at "Oct-28-2007 01:30 PST". After the upgrade to version 4, Job A is scheduled at "Oct-28-2007 01:45 PDT" and Job B remains at "Oct-28-2007 01:30 PDT", resulting in Job B being executed before Job A. Even though unchained scheduled jobs are not guaranteed to be executed in a certain order, this issue should be kept in mind when setting up scheduled jobs.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information regarding how to use these parameters

Clients and Servers Operating with Different Versions of Time Zone Files

In Oracle Database 11*g*, Release 11.2 (or higher), you can use different versions of time zone file on the client and server; this mode of operation was not supported prior to 11.2. Both client and server must be 11.2 or higher to operate in such a mixed mode. For the ramifications of working in such a mode, see *Oracle Call Interface Programmer's Guide*.

OCI, JDBC, Pro*C, and SQL*Plus clients can now continue to communicate with the database server without having to update client-side time zone files. For other products, if not explicitly stated in the product-specific documentation, it should be assumed that such clients cannot operate with a database server with a different time zone file than the client. Otherwise, computations on the TIMESTAMP WITH TIMEZONE values that are region ID based may give inconsistent results on the client. This is due to different daylight saving time (DST) rules in effect for the time zone regions affected between the different time zone file versions at the client and on the server.

Note if an application connects to different databases directly or via database links, it is recommended that all databases be on the same time zone file version. Otherwise, computations on the TIMESTAMP WITH TIMEZONE values on these different databases may give inconsistent results. This is due to different DST rules in effect for the time zone regions affected between the different time zone file versions across the different database servers.

Setting the Database Time Zone

Set the database time zone when the database is created by using the SET TIME_ZONE clause of the CREATE DATABASE statement. If you do not set the database time zone, then it defaults to the time zone of the server's operating system.

The time zone may be set to a named region or an absolute offset from UTC. To set the time zone to a named region, use a statement similar to the following example:

```
CREATE DATABASE db01
...
SET TIME_ZONE='Europe/London';
```

To set the time zone to an offset from UTC, use a statement similar to the following example:

```
CREATE DATABASE db01
```

```
...
SET TIME_ZONE='-05:00';
```

The range of valid offsets is -12:00 to +14:00.

> **Note:** The database time zone is relevant only for `TIMESTAMP WITH LOCAL TIME ZONE` columns. Oracle recommends that you set the database time zone to UTC (0:00) to avoid data conversion and improve performance when data is transferred among databases. This is especially important for distributed databases, replication, and exporting and importing.

You can change the database time zone by using the `SET TIME_ZONE` clause of the `ALTER DATABASE` statement. For example:

```
ALTER DATABASE SET TIME_ZONE='Europe/London';
ALTER DATABASE SET TIME_ZONE='-05:00';
```

The `ALTER DATABASE SET TIME_ZONE` statement returns an error if the database contains a table with a `TIMESTAMP WITH LOCAL TIME ZONE` column and the column contains data.

The change does not take effect until the database has been shut down and restarted.

You can find out the database time zone by entering the following query:

```
SELECT dbtimezone FROM DUAL;
```

Setting the Session Time Zone

You can set the default session time zone with the `ORA_SDTZ` environment variable. When users retrieve `TIMESTAMP WITH LOCAL TIME ZONE` data, Oracle Database returns it in the users' session time zone. The session time zone also takes effect when a `TIMESTAMP` value is converted to the `TIMESTAMP WITH TIME ZONE` or `TIMESTAMP WITH LOCAL TIME ZONE` data type.

> **Note:** Setting the session time zone does not affect the value returned by the `SYSDATE` and `SYSTIMESTAMP` SQL function. `SYSDATE` returns the date and time of the operating system on which the database resides, taking into account the time zone of the database server's operating system that was in effect when the database was started.

The `ORA_SDTZ` environment variable can be set to the following values:

- Operating system local time zone (`'OS_TZ'`)
- Database time zone (`'DB_TZ'`)
- Absolute offset from UTC (for example, `'-05:00'`)
- Time zone region name (for example, `'Europe/London'`)

To set `ORA_SDTZ`, use statements similar to one of the following in a UNIX environment (C shell):

```
% setenv ORA_SDTZ 'OS_TZ'
% setenv ORA_SDTZ 'DB_TZ'
```

```
% setenv ORA_SDTZ 'Europe/London'
% setenv ORA_SDTZ '-05:00'
```

When setting the ORA_SDTZ variable in a Microsoft Windows environment -- in the Registry, among system environment variables, or in a command prompt window -- do not enclose the time zone value in quotes.

You can change the time zone for a specific SQL session with the SET TIME_ZONE clause of the ALTER SESSION statement.

TIME_ZONE can be set to the following values:

- Default local time zone when the session was started (local)
- Database time zone (dbtimezone)
- Absolute offset from UTC (for example, '+10:00')
- Time zone region name (for example, 'Asia/Hong_Kong')

Use ALTER SESSION statements similar to the following:

```
ALTER SESSION SET TIME_ZONE=local;
ALTER SESSION SET TIME_ZONE=dbtimezone;
ALTER SESSION SET TIME_ZONE='Asia/Hong_Kong';
ALTER SESSION SET TIME_ZONE='+10:00';
```

You can find out the current session time zone by entering the following query:

```
SELECT sessiontimezone FROM DUAL;
```

Converting Time Zones With the AT TIME ZONE Clause

A datetime SQL expression can be one of the following:

- A datetime column
- A compound expression that yields a datetime value

A datetime expression can include an AT LOCAL clause or an AT TIME ZONE clause. If you include an AT LOCAL clause, then the result is returned in the current session time zone. If you include the AT TIME ZONE clause, then use one of the following settings with the clause:

- Time zone offset: The string '(+|-)HH:MM' specifies a time zone as an offset from UTC. For example, '-07:00' specifies the time zone that is 7 hours behind UTC. For example, if the UTC time is 11:00 a.m., then the time in the '-07:00' time zone is 4:00 a.m.
- DBTIMEZONE: Oracle Database uses the database time zone established (explicitly or by default) during database creation.
- SESSIONTIMEZONE: Oracle Database uses the session time zone established by default or in the most recent ALTER SESSION statement.
- Time zone region name: Oracle Database returns the value in the time zone indicated by the time zone region name. For example, you can specify Asia/Hong_Kong.
- An expression: If an expression returns a character string with a valid time zone format, then Oracle Database returns the input in that time zone. Otherwise, Oracle Database returns an error.

The following example converts the datetime value in the America/New_York time zone to the datetime value in the America/Los_Angeles time zone.

Example 4–5 Converting a Datetime Value to Another Time Zone

```
SELECT FROM_TZ(CAST(TO_DATE('1999-12-01 11:00:00',
     'YYYY-MM-DD HH:MI:SS') AS TIMESTAMP), 'America/New_York')
     AT TIME ZONE 'America/Los_Angeles' "West Coast Time"
FROM DUAL;

West Coast Time
---------------------------------------------------------
01-DEC-99 08.00.00.000000 AM AMERICA/LOS_ANGELES
```

See Also: *Oracle Database SQL Language Reference*

Support for Daylight Saving Time

Oracle Database automatically determines whether Daylight Saving Time is in effect for a specified time zone and returns the corresponding local time. Normally, date/time values are sufficient to allow Oracle Database to determine whether Daylight Saving Time is in effect for a specified time zone. The periods when Daylight Saving Time begins or ends are boundary cases. For example, in the Eastern region of the United States, the time changes from 01:59:59 a.m. to 3:00:00 a.m. when Daylight Saving Time goes into effect. The interval between 02:00:00 and 02:59:59 a.m. does not exist. Values in that interval are invalid. When Daylight Saving Time ends, the time changes from 02:00:00 a.m. to 01:00:01 a.m. The interval between 01:00:01 and 02:00:00 a.m. is repeated. Values from that interval are ambiguous because they occur twice.

To resolve these boundary cases, Oracle Database uses the `TZR` and `TZD` format elements. `TZR` represents the time zone region in datetime input strings. Examples are '`Australia/North`', '`UTC`', and '`Singapore`'. `TZD` represents an abbreviated form of the time zone region with Daylight Saving Time information. Examples are '`PST`' for U. S. Pacific Standard Time and '`PDT`' for U. S. Pacific Daylight Time. To see a list of valid values for the `TZR` and `TZD` format elements, query the `TZNAME` and `TZABBREV` columns of the `V$TIMEZONE_NAMES` dynamic performance view.

See Also:

- *Oracle Database SQL Language Reference* for more information regarding the session parameter `ERROR_ON_OVERLAP_TIME`
- "Time Zone Region Names" on page A-25 for a list of valid time zones

Examples: The Effect of Daylight Saving Time on Datetime Calculations

The `TIMESTAMP` data type does not accept time zone values and does not calculate Daylight Saving Time.

The `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` data types have the following behavior:

- If a time zone region is associated with the datetime value, then the database server knows the Daylight Saving Time rules for the region and uses the rules in calculations.
- Daylight Saving Time is not calculated for regions that do not use Daylight Saving Time.

The rest of this section contains examples that use datetime data types. The examples use the `global_orders` table. It contains the `orderdate1` column of `TIMESTAMP` data

type and the orderdate2 column of TIMESTAMP WITH TIME ZONE data type. The global_orders table is created as follows:

```
CREATE TABLE global_orders ( orderdate1 TIMESTAMP(0),
                        orderdate2 TIMESTAMP(0) WITH TIME ZONE);
INSERT INTO global_orders VALUES ( '28-OCT-00 11:24:54 PM',
                             '28-OCT-00 11:24:54 PM America/New_York');
```

Example 4–6 Comparing Daylight Saving Time Calculations Using TIMESTAMP WITH TIME ZONE and TIMESTAMP

```
SELECT orderdate1 + INTERVAL '8' HOUR, orderdate2 + INTERVAL '8' HOUR
       FROM global_orders;
```

The following output results:

```
ORDERDATE1+INTERVAL'8'HOUR      ORDERDATE2+INTERVAL'8'HOUR
--------------------------      --------------------------
29-OCT-00 07.24.54.000000 AM    29-OCT-00 06.24.54.000000 AM AMERICA/NEW_YORK
```

This example shows the effect of adding 8 hours to the columns. The time period includes a Daylight Saving Time boundary (a change from Daylight Saving Time to standard time). The orderdate1 column is of TIMESTAMP data type, which does not use Daylight Saving Time information and thus does not adjust for the change that took place in the 8-hour interval. The TIMESTAMP WITH TIME ZONE data type does adjust for the change, so the orderdate2 column shows the time as one hour earlier than the time shown in the orderdate1 column.

> **Note:** If you have created a global_orders table for the previous examples, then drop the global_orders table before you try Example 4–7 through Example 4–8.

Example 4–7 Comparing Daylight Saving Time Calculations Using TIMESTAMP WITH LOCAL TIME ZONE and TIMESTAMP

The TIMESTAMP WITH LOCAL TIME ZONE data type uses the value of TIME_ZONE that is set for the session environment. The following statements set the value of the TIME_ZONE session parameter and create a global_orders table. The global_orders table has one column of TIMESTAMP data type and one column of TIMESTAMP WITH LOCAL TIME ZONE data type.

```
ALTER SESSION SET TIME_ZONE='America/New_York';
CREATE TABLE global_orders ( orderdate1 TIMESTAMP(0),
                             orderdate2 TIMESTAMP(0) WITH LOCAL TIME ZONE );
INSERT INTO global_orders VALUES ( '28-OCT-00 11:24:54 PM',
                                   '28-OCT-00 11:24:54 PM' );
```

Add 8 hours to both columns.

```
SELECT orderdate1 + INTERVAL '8' HOUR, orderdate2 + INTERVAL '8' HOUR
FROM global_orders;
```

Because a time zone region is associated with the datetime value for orderdate2, the Oracle Database server uses the Daylight Saving Time rules for the region. Thus the output is the same as in Example 4–6. There is a one-hour difference between the two calculations because Daylight Saving Time is not calculated for the TIMESTAMP data type, and the calculation crosses a Daylight Saving Time boundary.

Example 4–8 Daylight Saving Time Is Not Calculated for Regions That Do Not Use Daylight Saving Time

Set the time zone region to UTC. UTC does not use Daylight Saving Time.

```
ALTER SESSION SET TIME_ZONE='UTC';
```

Truncate the `global_orders` table.

```
TRUNCATE TABLE global_orders;
```

Insert values into the `global_orders` table.

```
INSERT INTO global_orders VALUES ( '28-OCT-00 11:24:54 PM',
                                   TIMESTAMP '2000-10-28 23:24:54 ' );
```

Add 8 hours to the columns.

```
SELECT orderdate1 + INTERVAL '8' HOUR, orderdate2 + INTERVAL '8' HOUR
FROM global_orders;
```

The following output results.

```
ORDERDATE1+INTERVAL'8'HOUR                ORDERDATE2+INTERVAL'8'HOUR
--------------------------                --------------------------
29-OCT-00 07.24.54.000000000 AM           29-OCT-00 07.24.54.000000000 AM UTC
```

The times are the same because Daylight Saving Time is not calculated for the UTC time zone region.

5
Linguistic Sorting and Matching

This chapter explains linguistic sorting and searching for strings in an Oracle Database environment. The process of determining the mutual ordering of strings (character values) is called a collation. For any two strings, the collation defines whether the strings are equal or whether one precedes the other in the sorting order. In the Oracle documentation, the term sort is often used in place of *collation*.

Determining equality is especially important when a set of strings, such as a table column, is searched for values that equal a specified search term or that match a search pattern. SQL operators and functions used in searching are `=`, `LIKE`, `REGEXP_LIKE`, `INSTR`, and `REGEXP_INSTR`. This chapter uses the term *matching* to mean determining the equality of entire strings using the equality operator `=` or determining the equality of substrings of a string when the string is matched against a pattern using `LIKE`, `REGEXP_LIKE` or `REGEXP_INSTR`. Note that Oracle Text provides advanced full-text searching capabilities for the Oracle Database.

The ordering of strings in a set is called *sorting*. For example, the `ORDER BY` clause uses collation to determine the ordering of strings to sort the query results, while PL/SQL uses collations to sort strings in associative arrays indexed by `VARCHAR2` values, and the functions `MIN`, `MAX`, `GREATEST`, and `LEAST` use collations to find the smallest or largest character value.

There are many possible collations that can be applied to strings to determine their ordering. Collations that take into consideration the standards and customs of spoken languages are called linguistic collations. They order strings in the same way as dictionaries, phone directories, and other text lists written in a given language. In contrast, binary collation orders strings based on their binary representation (character encoding), treating each string as a simple sequences of bytes.

See Also:

- *Oracle Text Application Developer's Guide*

This chapter contains the following topics:

- Searching Linguistic Strings
- SQL Regular Expressions in a Multilingual Environment

Overview of Oracle Database Collation Capabilities

Different languages have different collations. 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.

Collation 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.

Collation 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`.

Collation order can be phonetic or it can be based on the appearance of the character. For example, collation can be based on the number of strokes in East Asian ideographs. Another common collation 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 Database has checked whether the next letter is an `h`.

Oracle Database provides the following types of collation:

- Binary
- Monolingual
- Multilingual
- Unicode Collation Algorithm (UCA)

While monolingual collation achieves a linguistically correct order for a single language, multilingual collation and UCA collation are designed to handle many languages at the same time. Furthermore, UCA collation conforms to the Unicode Collation Algorithm (UCA) that is a Unicode standard and is fully compatible with the international collation standard ISO 14651. The UCA standard provides a complete linguistic ordering for all characters in Unicode, hence all the languages around the world. With wide deployment of Unicode application, UCA collation is best suited for sorting multilingual data.

Using Binary Collation

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 collation**. Binary collation is the fastest type of sort. It produces 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 collation 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 collation is not usually linguistically meaningful for Asian languages that use ideographic characters.

Using Linguistic Collation

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

This section includes the following topics:

- Monolingual Collation
- Multilingual Collation
- UCA Collation

Monolingual Collation

Oracle Database compares character strings in two steps for monolingual collation. 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 Database. Oracle Database 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 5–1 illustrates sample values for sorting `a`, `A`, `ä`, `Ä`, and `b`.

Table 5–1 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

Note: Monolingual collation is not available for non-Unicode multibyte database character sets. If a monolingual collation is specified when the database character set is non-Unicode multibyte, then the default sort order is the binary sort order of the database character set. One exception is `UNICODE_BINARY`. This collation is available for all character sets.

See Also: "What is the Unicode Standard?" on page 6-1

Multilingual Collation

Oracle Database provides multilingual collation 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. As of Oracle Database 12*c*, Oracle Database supports all of the collations defined by previous releases.

For Asian language data or multilingual data, Oracle Database provides a sorting mechanism based on the ISO 14651 standard. For example, Chinese characters can be ordered by the number of strokes, PinYin, or radicals.

In addition, multilingual collation 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 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, Oracle Database supports a monolingual French sort (`FRENCH`), but you can specify a multilingual French collation (`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. Multilingual linguistic sort is usually used if the tables contain multilingual data. If the tables contain only French, then a monolingual French sort might 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 trade-off between the scope and the performance of a sort.

See Also:

- "Canonical Equivalence" on page 5-12
- "Code Points and Supplementary Characters" on page 6-2

Multilingual Collation Levels

Oracle Database evaluates multilingual collation at three levels of precision:

- Primary Level Collation
- Secondary Level Collation
- Tertiary Level Collation

Primary Level Collation A primary level collation distinguishes between **base letters**, such as the difference between characters `a` and `b`. It is up to individual locales to define whether `a` is before `b`, `b` is before `a`, or if 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 5-8

Secondary Level Collation A secondary level collation distinguishes between base letters (the primary level collation) before distinguishing between diacritics on a given base letter. 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 letter (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 Collation A tertiary level collation distinguishes between base letters (primary level collation), diacritics (secondary level collation), and case (upper case and lower case). It can also include special characters such as +, -, and *.

The following are examples of tertiary level collations:

- 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 weight 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 Database 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
Resumes
résumés
Résumés
```

UCA Collation

The Unicode Collation Algorithm (UCA) is a Unicode standard that is fully compatible with the international collation standard ISO 14651. The UCA defines a Default Unicode Collation Element Table (DUCET) that provides a reasonable default ordering for all languages that are not tailored. To achieve the correct ordering for a particular language, DUCET can be tailored to meet the linguistic requirements for that language. There are tailorings of DUCET for various languages provided in the Unicode Common Locale Data Repository. For details of the UCA and related terminologies, see the Unicode Collation Algorithm at http://www.unicode.org.

Oracle Database provides UCA collation that fully conforms to the UCA 6.2.0 as of Oracle Database 12*c* (12.1.0.2). In addition to the collation based on DUCET, it provides tailored collations for a number of commonly used languages. For example, you can specify the UCA collation, `UCA0620_SCHINESE`, to sort multilingual data containing Simplified Chinese. The collation will make Simplified Chinese data appear in the PinYin order.

For sorting multilingual data, Oracle Corporation recommends UCA collation.

UCA Comparison Levels

Similar to multilingual collation, UCA collations employ a multilevel comparison algorithm to evaluate characters. This can go up to four levels of comparison:

- Primary Level
- Secondary Level
- Tertiary Level
- Quaternary Level

Primary Level The primary level is used to distinguish between base letters, which is similar to the comparison used in the primary level collation of the multilingual collation.

> **See Also:** "Primary Level Collation" on page 5-4 for examples of base letter differences

Secondary Level The secondary level is used to distinguish between diacritics if base letters are the same, which is similar to what is used in the secondary level collation of the multilingual collation to distinguish between diacritics.

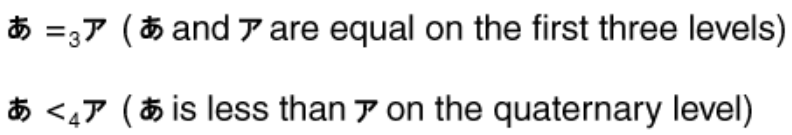
> **See Also:** "Secondary Level Collation" on page 5-5 for examples of diacritic differences

Tertiary Level The tertiary level is used to distinguish between cases on a given base letter with the same diacritic, which is similar to what is used in the tertiary level collation of the multilingual collation to distinguish between cases. Moreover, UCA DUCET collation treats punctuations with primary or quaternary significance based on how variable characters are weighted, which is different from the tertiary level collation of the multilingual collation that treat punctuations with tertiary level of significance.

> **See Also:** "Tertiary Level Collation" on page 5-5 for examples of characters with case differences

Quaternary Level The quaternary level is used to distinguish variable characters from other characters if variable characters are weighted as shifted. It is also used to distinguish Hiragana from Katakana with the same base and case. An example is illustrated in Figure 5–1.

Figure 5–1 Hiragana and Katakana Collation



See Also: "UCA Collation Parameters" on page 5-7

UCA Collation Parameters Table 5–2 illustrates the collation parameters and options that are supported in UCA collations in Oracle Database 12*c*.

Table 5–2 UCA Collation Parameters

Attribute	Options	Collation Modifier
`strength`	primary	`_AI` or `_S1`
	secondary	`_CI` or `_S2`
	tertiary	`_S3`
	quaternary	`_S4`[1]
`alternate`	non-ignorable	`_VN`
	shifted	`_VS`
	blanked	`_VB`
`backwards`	on	`_BY`
	off	`_BN`
`normalization`	on	`_NY`
`caseLevel`	off	`_EN`
`caseFirst`	upper	`_FU`[2]
	off	`_FN`[3]
`hiraganaQuaternary`	on	`_HY`
	off	`_HN`
`numeric`	off	`_DN`
`match-style`	minimal	`_MN`

[1] `_S4`: Applicable only when alternate is shifted
[2] `_FU`: Only valid for Danish
[3] `_FN`: Only valid for other languages

The parameter `strength` represents UCA comparison level (see "UCA Comparison Levels" on page 5-6). The parameter `alternate` controls how variable characters are weighted (see "Variable Characters and Variable Weighting" on page 5-9). The parameter `backwards` controls if diacritics are to be sorted backward (see "Reverse Secondary Sorting" on page 5-12). Users can configure these three UCA parameters using the options listed in Table 5–2. The options for the other parameters listed in Table 5–2 are currently fixed based on tailored languages and not configurable as of Oracle Database 12*c*.

For a complete description of UCA collation parameters and options, see the Unicode Collation Algorithm standard at: `http://www.unicode.org`.

Linguistic Collation Features

This section contains information about different features that a linguistic collation can 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 collations 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

In multilingual collation and UCA collation, certain characters may be treated as ignorable. **Ignorable characters** are skipped, that is, treated as non-existent, when two character values (strings) containing such characters are compared in a sorting or matching operation. There are three kinds of ignorable characters: primary, secondary, and tertiary.

- Primary Ignorable Characters
- Secondary Ignorable Characters
- Tertiary Ignorable Characters

Primary Ignorable Characters

Primary ignorable characters are ignored when the multilingual collation or UCA collation definition applied to the given comparison has the accent-insensitivity modifier `_AI`, for example, `GENERIC_M_AI` or `UCA0620_DUCET_AI`. Primary ignorable characters are comprised of diacritics (accents) from various alphabets (Latin, Cyrillic, Greek, Devanagari, Katakana, and so on) but also of decorating modifiers, such as an enclosing circle or enclosing square. These characters are non-spacing combining characters, which means they combine with the preceding character to form a complete accented or decorated character ("non-spacing" means that the character occupies the same character position on screen or paper as the preceding character). For example, the character "Latin Small Letter e" followed by the character "Combining Grave Accent" forms a single letter "è", while the character "Latin Capital Letter A" followed by the "Combining Enclosing Circle" forms a single character "(A)". Because non-spacing characters are defined as ignorable for accent-insensitive sorts, these sorts can treat, for example, `rôle` as equal to `role`, `naïve` as equal to `naive`, and `(A)(B)(C)` as equal to `ABC`.

Primary ignorable characters are called non-spacing characters when viewed in a multilingual collation definition in the Oracle Locale Builder utility.

Secondary Ignorable Characters

Secondary ignorable characters are ignored when the applied definition has either the accent-insensitivity modifier `_AI` or the case-insensitivity modifier `_CI`.

In multilingual collation, secondary ignorable characters are comprised of punctuation characters, such as the space character, new line control codes, dashes, various quote forms, mathematical operators, dot, comma, exclamation mark, various bracket forms, and so on. In accent-insensitive (`_AI`) and case-insensitive (`_CI`) sorts, these punctuation characters are ignored so that `multi-lingual` can be treated as equal to `multilingual` and `e-mail` can be treated as equal to `email`.

Secondary ignorable characters are called punctuation characters when viewed in a multilingual collation definition in the Oracle Locale Builder utility.

There are no secondary ignorable characters defined in the UCA DUCET, however. Punctuations are treated as variable characters in the UCA.

Tertiary Ignorable Characters

Tertiary ignorable characters are generally ignored in linguistic comparison. They are mainly comprised of control codes, format characters, variation selectors, and so on.

Primary and secondary ignorable characters are not ignored when a standard, case- and accent-sensitive sort is used. However, they have lower priority when determining the order of strings. For example, `multi-lingual` is sorted after `multilingual` in the `GENERIC_M` sort, but it is still sorted between `multidimensional` and `multinational`. The comparison `d < l < n` of the base letters has higher priority in determining the order than the presence of the secondary ignorable character HYPHEN (U+002D).

You can see the full list of non-spacing characters and punctuation characters in a multilingual collation definition when viewing the definition in the Oracle Locale Builder. Generally, neither punctuation characters nor non-spacing characters are included in monolingual collation definitions. In some monolingual collation definitions, the space character and the tabulator character may be included. The comparison algorithm automatically assigns a minor value to each undefined character. This makes punctuation characters non-ignorable but, as in the case of multilingual collations, considered with lower priority when determining the order of compared strings. The ordering among punctuation characters in monolingual collations is based on their Unicode code points and may not correspond to user expectations.

See Also: "Case-Insensitive and Accent-Insensitive Linguistic Collation" on page 5-14

Variable Characters and Variable Weighting

There are characters defined with variable collation elements in the UCA. These characters are called variable characters and are comprised of white space characters, punctuations, and certain symbols.

Variable characters can be weighted differently in UCA collations to adjust the effect of these characters in a sorting or comparison, which is called variable weighting. The collation parameter, `alternate`, controls how it works. The following options on variable weighting are supported in UCA collations as of Oracle Database 12*c*:

- `blanked`

  Variable characters are treated as ignorable characters. For example, SPACE (U+0020) is ignored in comparison.

- `non-ignorable`

  Variable characters are treated as if they were not ignorable characters. For example, SPACE (U+0020) is not ignored in comparison at primary level.

- `shifted`

  Variable characters are treated as ignorable characters on the primary, secondary and tertiary levels. In addition, a new quaternary level is used for all characters. The quaternary weight of a character depends on if the character is a variable, ignorable, or other. For example, SPACE (U+0020) is assigned a quaternary weight differently from A (U+0041) because SPACE is a variable character while A is neither a variable nor an ignorable character.

See Also: "UCA Collation Parameters" on page 5-7

Examples of Variable Weighting

This section includes different examples of variable weighting.

Example 5–1 UCA DUCET Order When Variable is Weighed as Blanked

The following list has been sorted using `UCA0620_DUCET_VB`:

```
blackbird
Blackbird
Black-bird
Black bird
BlackBird
```

`Blackbird`, `Black-bird`, and `Black bird` have the same collation weight because SPACE(U+0020) and HYPHEN(U+002D) are treated as ignorable characters. Selecting only the distinct entries illustrates this behavior (note that only `Blackbird` is shown in the result):

```
blackbird
Blackbird
BlackBird
```

`Blackbird`, `Black-bird`, and `Black bird` are sorted after `blackbird` due to case difference on the first letter B (U+0042), but before `BlackBird` due to case difference at the second `b` (U+0062).

Example 5–2 UCA DUCET Order When Variable is Weighed as Non-Ignorable

The following list has been sorted using `UCA0620_DUCET_VN`:

```
Black bird
Black-bird
blackbird
Blackbird
BlackBird
```

`Black bird` and `Black-bird` are sorted before `blackbird` because both SPACE (U+0020) and HYPHEN (U+002D) are not treated as ignorable characters but they are smaller than `b` (U+0062) at the primary level. `Black bird` is sorted before `Black-bird` because SPACE (U+0020) is small than HYPHEN (U+002D) at the primary level.

Example 5–3 UCA DUCET Order When Variable is Weighed as Shifted

The following list has been sorted using `UCA0620_DUCET`:

```
blackbird
Black bird
Black-bird
Blackbird
BlackBird
```

`blackbird` is sorted before `Black bird` and `Black-bird` because both SPACE (U+0020) and HYPHEN (U+002D) are ignored at the first three levels, and there is a case difference on the first letter `b` (U+0062). `Black-bird` is sorted before `Blackbird` is because HYPHEN (U+002D) has a small quaternary weight than the letter `b` (U+0062) in `Blackbird`.

Contracting Characters

Collation 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 collation element during sorting. For example, in traditional Spanish, the string `ch` is composed of two characters. These characters are called **contracting characters** in multilingual collation and **special combination letters** in monolingual collation.

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 collation and **special letters** in monolingual collation. 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 collation. 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

Canonical equivalence is an attribute of a multilingual collation and describes how equivalent code point sequences are sorted. If canonical equivalence is applied in a particular multilingual collation, then canonically equivalent strings are treated as equal.

One Unicode code point can be equivalent to a sequence of base letter code points plus diacritic code points. 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 need to be applied in a specific multilingual collation. Oracle Database-defined multilingual collations include the appropriate setting for the canonical equivalence flag. You can set the flag to FALSE to speed up the comparison and ordering functions if all the data is in its composed form.

For example, consider the following strings:

- äa (a umlaut followed by a)
- a¨b (a followed by umlaut followed by b)
- äc (a umlaut followed by c)

If CANONICAL_EQUIVALENCE=FALSE, then the sort order of the strings is:

```
a¨b
äa
äc
```

This occurs because a comes before ä if canonical equivalence is not applied.

If CANONICAL_EQUIVALENCE=TRUE, then the sort order of the strings is:

```
äa
a¨b
äc
```

This occurs because ä and a¨ are treated as canonically equivalent.

You can use Oracle Locale Builder to view the setting of the canonical equivalence flag in existing multilingual collations. When you create a customized multilingual collation with Oracle Locale Builder, you can set the canonical equivalence flag as desired.

> **See Also:** "Creating a New Linguistic Sort with the Oracle Locale Builder" on page 12-25 for more information about setting the canonical equivalence flag

Reverse Secondary Sorting

In French, sorting strings of characters with diacritics first compares base letters 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 12-25 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 characters 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 12-25 for more information about setting the `SWAP_WITH_NEXT` flag

Special Letters

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

See Also: "Expanding Characters" on page 5-11

Special Combination Letters

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

See Also: "Contracting Characters" on page 5-11

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 collations. The `UPPER`, `LOWER`, and `INITCAP` SQL functions cannot handle these special characters, because their casing operation is based on binary mapping defined for the underlying character set, which is not linguistic sensitive.

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: *Oracle Database SQL Language Reference*

Special Lowercase Letters

Oracle Database 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.

Case-Insensitive and Accent-Insensitive Linguistic Collation

Operation inside an Oracle database is always sensitive to the case and the accents (diacritics) of the characters. Sometimes you may need to perform case-insensitive or accent-insensitive comparisons and collations.

In previous versions of the database, case-insensitive queries could be achieved by using the `NLS_UPPER` and `NLS_LOWER` SQL functions. The functions change the case of strings based on a specific linguistic collation definition. This enables you to perform case-insensitive searches regardless of the language being used. For example, create a table called `test1` as follows:

```
SQL> CREATE TABLE test1(word VARCHAR2(12));
SQL> INSERT INTO test1 VALUES('GROSSE');
SQL> INSERT INTO test1 VALUES('Große');
SQL> INSERT INTO test1 VALUES('große');
SQL> SELECT * FROM test1;

WORD
------------
GROSSE
Große
große
```

Perform a case-sensitive search for `GROSSE` as follows:

```
SQL> SELECT word FROM test1 WHERE word='GROSSE';

WORD
------------
GROSSE
```

Perform a case-insensitive search for `GROSSE` using the `NLS_UPPER` function:

```
SELECT word FROM test1
WHERE NLS_UPPER(word, 'NLS_SORT = XGERMAN') = 'GROSSE';

WORD
------------
GROSSE
Große
große
```

Oracle Database provides case-insensitive and accent-insensitive options for collation. It provides the following types of linguistic collations:

- Linguistic collations that use information about base letters, diacritics, punctuation, and case. These are the standard linguistic collations that are described in "Using Linguistic Collation" on page 5-3.
- Monolingual collations that use information about base letters, diacritics, and punctuation, but not case, and multilingual and UCA collations that use information about base letters and diacritics, but not case or punctuation. This type of sort is called **case-insensitive**.
- Monolingual collations that use information about base letters and punctuation only, and multilingual and UCA collations that use information about base letters only. This type of sort is called **accent-insensitive**. (**Accent** is another word for **diacritic**.) Like case-insensitive sorts, an accent-insensitive sort does not use information about case.

Accent- and case-insensitive multilingual collations ignore punctuation characters as described in "Ignorable Characters" on page 5-8.

The rest of this section contains the following topics:

- Examples: Case-Insensitive and Accent-Insensitive Collation
- Specifying a Case-Insensitive or Accent-Insensitive Collation
- Examples: Linguistic Collation

See Also:

- "NLS_SORT" on page 3-30
- "NLS_COMP" on page 3-31

Examples: Case-Insensitive and Accent-Insensitive Collation

The following examples show:

- A collation that uses information about base letters, diacritics, punctuation, and case
- A case-insensitive collation
- An accent-insensitive collation

Example 5–4 Linguistic Collation Using Base Letters, Diacritics, Punctuation, and Case Information

The following list has been sorted using information about base letters, diacritics, punctuation, and case:

```
blackbird
black bird
black-bird
Blackbird
Black-bird
blackbîrd
bläckbird
```

Example 5–5 Case-Insensitive Linguistic Collation

The following list has been sorted using information about base letters, diacritics, and punctuation, ignoring case:

```
black bird
black-bird
Black-bird
blackbird
Blackbird
blackbîrd
bläckbird
```

`black-bird` and `Black-bird` have the same value in the collation, because the only different between them is case. They could appear interchanged in the list. `Blackbird` and `blackbird` also have the same value in the collation and could appear interchanged in the list.

Example 5–6 Accent-Insensitive Linguistic Collation

The following list has been sorted using information about base letters only. No information about diacritics, punctuation, or case has been used.

```
blackbird
bläckbird
blackbîrd
Blackbird
BlackBird
Black-bird
Black bird
```

Specifying a Case-Insensitive or Accent-Insensitive Collation

Use the NLS_SORT session parameter to specify a case-insensitive or accent-insensitive collation:

- Append _CI to an Oracle Database collation name for a case-insensitive collation.
- Append _AI to an Oracle Database collation name for an accent-insensitive and case-insensitive collation.

For example, you can set NLS_SORT to the following types of values:

```
UCA0620_SPANISH_AI
FRENCH_M_AI
XGERMAN_CI
```

Binary collation can also be case-insensitive or accent-insensitive. When you specify BINARY_CI as a value for NLS_SORT, it designates a collation that is accent-sensitive and case-insensitive. BINARY_AI designates an accent-insensitive and case-insensitive binary collation. You may want to use a binary collation if the binary collation order of the character set is appropriate for the character set you are using.

For example, with the NLS_LANG environment variable set to AMERICAN_AMERICA.WE8ISO8859P1, create a table called test2 and populate it as follows:

```
SQL> CREATE TABLE test2 (letter VARCHAR2(10));
SQL> INSERT INTO test2 VALUES('ä');
SQL> INSERT INTO test2 VALUES('a');
SQL> INSERT INTO test2 VALUES('A');
SQL> INSERT INTO test2 VALUES('Z');
SQL> SELECT * FROM test2;

LETTER
-----------
ä
a
A
Z
```

The default value of NLS_SORT is BINARY. Use the following statement to do a binary collation of the characters in table test2:

```
SELECT * FROM test2 ORDER BY letter;
```

To change the value of NLS_SORT, enter a statement similar to the following:

```
ALTER SESSION SET NLS_SORT=BINARY_CI;
```

The following table shows the collation orders that result from setting NLS_SORT to BINARY, BINARY_CI, and BINARY_AI.

BINARY	BINARY_CI	BINARY_AI
A	a	ä

BINARY	BINARY_CI	BINARY_AI
Z	A	a
a	Z	A
ä	ä	Z

When NLS_SORT=BINARY, uppercase letters come before lowercase letters. Letters with diacritics appear last.

When the collation considers diacritics but ignores case (BINARY_CI), the letters with diacritics appear last.

When both case and diacritics are ignored (BINARY_AI), ä is sorted with the other characters whose base letter is a. All the characters whose base letter is a occur before z.

You can use binary collation for better performance when the character set is US7ASCII or another character set that has the same collation order as the binary collation.

The following table shows the collation orders that result from German collation for the table.

GERMAN	GERMAN_CI	GERMAN_AI
a	a	ä
A	A	a
ä	ä	A
Z	Z	Z

A German collation places lowercase letters before uppercase letters, and ä occurs before Z. When the collation ignores both case and diacritics (GERMAN_AI), ä appears with the other characters whose base letter is a.

Examples: Linguistic Collation

The examples in this section demonstrate a binary collation, a monolingual collation, and a UCA collation. To prepare for the examples, create and populate a table called test3. Enter the following statements:

```
SQL> CREATE TABLE test3 (name VARCHAR2(20));
SQL> INSERT INTO test3 VALUES('Diet');
SQL> INSERT INTO test3 VALUES('À voir');
SQL> INSERT INTO test3 VALUES('Freizeit');
```

Example 5–7 Binary Collation

The ORDER BY clause uses a binary collation.

```
SQL> SELECT * FROM test3 ORDER BY name;
```

You should see the following output:

```
Diet
Freizeit
À voir
```

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

Example 5–8 Monolingual German Collation

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

```
SQL> SELECT * FROM test3 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 collation.

Example 5–9 Comparing a Monolingual German Collation to a UCA Collation

Insert the character string shown in Figure 5–2 into `test`. It is a `D` with a crossbar followed by `ñ`.

Figure 5–2 Character String

Đñ

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

```
SELECT * FROM test2 ORDER BY NLSSORT(name, 'NLS_SORT=german');
```

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

Perform a UCA collation by entering the following statement:

```
SELECT * FROM test2
ORDER BY NLSSORT(name, 'NLS_SORT=UCA0620_DUCET');
```

The output shows the new character string after `Diet`, following the UCA order.

See Also:

- "The NLSSORT Function" on page 9-7
- "NLS_SORT" on page 3-30 for more information about setting and changing the `NLS_SORT` parameter

Performing Linguistic Comparisons

When performing SQL comparison operations, characters are compared according to their binary values. A character is greater than another if it has a higher binary value. Because the binary sequences rarely match the linguistic sequences for most languages, such comparisons may not be meaningful for a typical user. To achieve a meaningful comparison, you can specify behavior by using the session parameters `NLS_COMP` and `NLS_SORT`. The way you set these two parameters determines the rules by which characters are collated and compared.

The `NLS_COMP` setting determines how `NLS_SORT` is handled by the SQL operations. There are three valid values for `NLS_COMP`:

- BINARY

  All SQL collations and comparisons are based on the binary values of the string characters, regardless of the value set to NLS_SORT. This is the default setting.

- LINGUISTIC

  All SQL collation and comparison are based on the linguistic rule specified by NLS_SORT. For example, NLS_COMP=LINGUISTIC and NLS_SORT=BINARY_CI means the collation sensitive SQL operations will use binary value for collating and comparison but ignore character case.

- ANSI

  A limited set of SQL functions honor the NLS_SORT setting. ANSI is available for backward compatibility only. In general, you should set NLS_COMP to LINGUISTIC when performing linguistic comparison.

Table 5–3 shows how different SQL or PL/SQL operations behave with these different settings.

Table 5–3 Linguistic Comparison Behavior with NLS_COMP Settings

SQL or PL/SQL Operation	BINARY	LINGUISTIC	ANSI
Set Operators	-	-	-
UNION, INTERSECT, MINUS	Binary	Honors NLS_SORT	Binary
Scalar Functions	-	-	-
DECODE	Binary	Honors NLS_SORT	Binary
INSTRx	Binary	Honors NLS_SORT	Binary
LEAST, GREATEST	Binary	Honors NLS_SORT	Binary
MAX, MIN	Binary	Honors NLS_SORT	Binary
NLS_INITCAP	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
NLS_LOWER	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
NLS_UPPER	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
NLSSORT	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
NULLIF	Binary	Honors NLS_SORT	Binary
REGEXP_COUNT	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
REGEXP_INSTR	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
REGEXP_REPLACE	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
REGEXP_SUBSTR	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
REPLACE	Binary	Honors NLS_SORT	Binary
RTRIM, TRIM, LTRIM	Binary	Honors NLS_SORT	Binary
TRANSLATE	Binary	Honors NLS_SORT	Binary
Conditions	-	-	-
=, !=, >, <, >=, <=	Binary	Honors NLS_SORT	Honors NLS_SORT
BETWEEN, NOT BETWEEN	Binary	Honors NLS_SORT	Honors NLS_SORT
IN, NOT IN	Binary	Honors NLS_SORT	Honors NLS_SORT
LIKE	Binary	Honors NLS_SORT	Binary

Table 5–3 (Cont.) Linguistic Comparison Behavior with NLS_COMP Settings

SQL or PL/SQL Operation	BINARY	LINGUISTIC	ANSI
REGEXP_LIKE	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
CASE Expression	-	-	-
CASE	Binary	Honors NLS_SORT	Binary
Analytic Function Clauses	-	-	-
DISTINCT	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
OVER(ORDER BY)	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
OVER(PARTITION BY)	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT
Subquery Clauses	-	-	-
DISTINCT, UNIQUE	Binary	Honors NLS_SORT	Binary
GROUP BY	Binary	Honors NLS_SORT	Binary
ORDER BY	Honors NLS_SORT	Honors NLS_SORT	Honors NLS_SORT

See "NLS_COMP" and "NLS_SORT" for information regarding these parameters.

Collation Keys

When the comparison conditions =, !=, >, <, >=, <=, BETWEEN, NOT BETWEEN, IN, NOT IN, the query clauses ORDER BY or GROUP BY, or the aggregate function COUNT(DISTINCT) are evaluated according to linguistic rules specified by NLS_SORT, the compared argument values are first transformed to binary values called collation keys and then compared byte by byte, like RAW values. If a monolingual collation is applied, collation keys contain major values for characters of the source value concatenated with minor values for those characters. If a multilingual collation is applied, collation keys contain concatenated primary, secondary, and tertiary values.

The collation keys are the same values that are returned by the NLSSORT function. That is, activating the linguistic behavior of these SQL operations is equivalent to including their arguments into calls to the NLSSORT function.

Restricted Precision of Linguistic Comparison

As collation keys are values of the data type RAW and the maximum length of a RAW value depends on the value of the initialization parameter, MAX_STRING_SIZE, the maximum length of a collation key is controlled by the parameter as well.

When MAX_STRING_SIZE is set to STANDARD, the maximum length of a collation key is restricted to 2000 bytes. If a full source string yields a collation key longer than the maximum length, the collation key generated for this string is calculated for a maximum prefix (initial substring) of the value for which the calculated result does not exceed 2000 bytes. For monolingual collation, the prefix is typically 1000 characters. For multilingual collation, the prefix is typically 500 characters. For UCA collations, the prefix is typically 300 characters. The exact length of the prefix may be higher or lower and depends on the particular collation and the particular characters contained in the source string. The implication of this method of collation key generation is that SQL operations using the collation keys to implement the linguistic behavior will return results that may ignore trailing parts of long arguments. For example, two strings starting with the same 1000 characters but differing somewhere after the 1000th character will be grouped together by the GROUP BY clause.

When MAX_STRING_SIZE is set to EXTENDED, the maximum length of a collation key is restricted to 32767 bytes. If a full source string yields a collation key longer than the maximum length, ORA- 12742 will be raised.

Examples: Linguistic Comparison

The following examples illustrate behavior with different NLS_COMP settings.

Example 5–10 Binary Comparison Binary Collation

The following illustrates behavior with a binary setting:

```
SQL> ALTER SESSION SET NLS_COMP=BINARY;
SQL> ALTER SESSION SET NLS_SORT=BINARY;
SQL> SELECT ename FROM emp1;

ENAME
----------------------
Mc Calla
MCAfee
McCoye
Mccathye
McCafeé

5 rows selected

SQL> SELECT ename FROM emp1 WHERE ename LIKE 'McC%e';

ENAME
----------------------
McCoye

1 row selected
```

Example 5–11 Linguistic Comparison Binary Case-Insensitive Collation

The following illustrates behavior with a case-insensitive setting:

```
SQL> ALTER SESSION SET NLS_COMP=LINGUISTIC;
SQL> ALTER SESSION SET NLS_SORT=BINARY_CI;
SQL> SELECT ename FROM emp1 WHERE ename LIKE 'McC%e';

ENAME
----------------------
McCoye
Mccathye

2 rows selected
```

Example 5–12 Linguistic Comparison Binary Accent-Insensitive Collation

The following illustrates behavior with an accent-insensitive setting:

```
SQL> ALTER SESSION SET NLS_COMP=LINGUISTIC;
SQL> ALTER SESSION SET NLS_SORT=BINARY_AI;
SQL> SELECT ename FROM emp1 WHERE ename LIKE 'McC%e';

ENAME
----------------------
McCoye
Mccathye
```

```
McCafeé

3 rows selected
```

Example 5–13 Linguistic Comparisons Returning Fewer Rows

Some operations may return fewer rows after applying linguistic rules. For example, with a binary setting, `McAfee` and `Mcafee` are different:

```
SQL> ALTER SESSION SET NLS_COMP=BINARY;
SQL> ALTER SESSION SET NLS_SORT=BINARY;
SQL> SELECT DISTINCT ename FROM emp2;

ENAME
----------------------
McAfee
Mcafee
McCoy

3 rows selected
```

However, with a case-insensitive setting, `McAfee` and `Mcafee` are the same:

```
SQL> ALTER SESSION SET NLS_COMP=LINGUISTIC;
SQL> ALTER SESSION SET NLS_SORT=BINARY_CI;
SQL> SELECT DISTINCT ename FROM emp2;

ENAME
----------------------
McAfee
McCoy

2 rows selected
```

In this example, either `McAfee` or `Mcafee` could be returned from the `DISTINCT` operation. There is no guarantee exactly which one will be picked.

Example 5–14 Linguistic Comparisons Using XSPANISH

There are cases where characters are the same using binary comparison but different using linguistic comparison. For example, with a binary setting, the character `C` in `Cindy`, `Chad`, and `Clara` represents the same letter `C`:

```
SQL> ALTER SESSION SET NLS_COMP=BINARY;
SQL> ALTER SESSION SET NLS_SORT=BINARY;
SQL> SELECT ename FROM emp3 WHERE ename LIKE 'C%';

ENAME
----------------------
Cindy
Chad
Clara

3 rows selected
```

In a database session with the linguistic rule set to traditional Spanish, `XSPANISH`, `ch` is treated as one character. So the letter `c` in `Chad` is different than the letter `C` in `Cindy` and `Clara`:

```
SQL> ALTER SESSION SET NLS_COMP=LINGUISTIC;
SQL> ALTER SESSION SET NLS_SORT=XSPANISH;
```

```
SQL> SELECT ename FROM emp3 WHERE ename LIKE 'C%';

ENAME
----------------------
Cindy
Clara

2 rows selected
```

And the letter c in combination ch is different than the c standing by itself:

```
SQL> SELECT REPLACE ('character', 'c', 't') "Changes" FROM DUAL;

Changes
---------------------
charatter
```

Example 5–15 Linguistic Comparisons Using UCA0620_TSPANISH

The character ch behaves the same in the traditional Spanish ordering of the UCA collations as that in XSPANISH:

```
SQL> ALTER SESSION SET NLS_COMP = LINGUISTIC;
SQL> ALTER SESSION SET NLS_SORT = UCA0620_TSPANISH;
SQL> SELECT ename FROM emp3 WHERE ename LIKE 'C%';

ENAME
--------------
Cindy
Clara

SQL> SELECT REPLACE ('character', 'c', 't') "Changes" FROM DUAL;

Changes
-----------
charatter
```

Using Linguistic Indexes

Linguistic collation is language-specific and requires more data processing than binary collation. Using a binary collation 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 collate the data returned from a SELECT...ORDER BY statement according to different collation 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 collation with the ORDER BY clause and the WHERE clause.

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

The following statement creates an index based on a German collation:

```
CREATE TABLE my_table(name VARCHAR(20) NOT NULL);
CREATE INDEX nls_index ON my_table (NLSSORT(name, 'NLS_SORT = German'));
```

The NOT NULL in the CREATE TABLE statement ensures that the index is used.

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

```
SELECT * FROM my_table ORDER BY name
WHERE name LIKE 'Hein%';
```

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

The rest of this section contains the following topics:

- Supported SQL Operations and Functions for Linguistic Indexes
- Linguistic Indexes for Multiple Languages
- Requirements for Using Linguistic Indexes

See Also:

- *Oracle Database Administrator's Guide* for more information about function-based indexes

Supported SQL Operations and Functions for Linguistic Indexes

Linguistic index support is available for the following collation-sensitive SQL operations and SQL functions:

- Comparison conditions =, !=, >, <, >=, <=
- Range conditions BETWEEN | NOT BETWEEN
- IN | NOT IN
- ORDER BY
- GROUP BY
- LIKE (LIKE, LIKE2, LIKE4, LIKEC)
- DISTINCT
- UNIQUE
- UNION
- INTERSECT
- MINUS

The SQL functions in the following list cannot utilize linguistic index:

- INSTR (INSTR, INSTRB, INSTR2, INSTR4, INSTRC)
- MAX
- MIN
- REPLACE
- TRIM
- LTRIM
- RTRIM
- TRANSLATE

Linguistic Indexes for Multiple Languages

There are four 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 Database 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`, then Oracle Database uses `french_index`. When it is set to `GERMAN`, Oracle Database 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 5-26) 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 collations 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 Collation" on page 5-4 for more information

- Build a single linguistic index for all languages using one of the UCA collations such as `UCA0620_DUCET` or `UCA0620_CFRENCH`. These indexes sort characters in the order conforming to ISO 14654 and UCA 6.2.0. For example:

  ```
  CREATE INDEX i
    ON t (NLSSORT(col, 'NLS_SORT=UCA0620_DUCET'));
  ```

  > **See Also:** "UCA Collation" on page 5-5 for more information

Requirements for Using Linguistic Indexes

The following are requirements for using linguistic indexes:

- Set NLS_SORT Appropriately
- Specify NOT NULL in a WHERE Clause If the Column Was Not Declared NOT NULL
- Use a Tablespace with an Adequate Block Size

This section also includes:

Set NLS_SORT Appropriately

The NLS_SORT parameter should indicate the linguistic definition you want to use for the linguistic collation. If you want a French linguistic collation order, then NLS_SORT should be set to FRENCH. If you want a German linguistic collation order, then 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 3-30

Specify NOT NULL in a WHERE Clause If the Column Was Not Declared NOT NULL

When you want to use the ORDER BY *column_name* clause with a column that has a linguistic index, include a WHERE clause like the following example:

```
WHERE NLSSORT(column_name) IS NOT NULL
```

This WHERE clause is not necessary if the column has already been defined as a NOT NULL column in the schema.

Use a Tablespace with an Adequate Block Size

A collation key created from a character value is usually a few times longer than this value. The actual length expansion depends on the particular collation in use and the content of the source value, with the UCA-based collations expanding the most.

When creating a linguistic index, Oracle Database first calculates the estimated maximum size of the index key by summing up the estimated maximum sizes of the collation keys (NLSSORT results) for each of the character columns forming the index key. In this calculation, the maximum size of a collation key for a character column with the maximum byte length n is estimated to be n*21+5 for UCA-based collations and n*8+10 for other collations.

The large expansion ratios can yield large maximum index key sizes, especially for composite (multicolumn) keys. At the same time, the maximum key size of an index cannot exceed around 70% of the block size of the tablespace containing the index. If it does, an ORA-1450 error is reported. To avoid this error, you should store the linguistic index in a tablespace with an adequate block size, which may be larger than the default block size of your database. A suitable tablespace can be created with the CREATE TABLESPACE statement, provided the initialization parameter DB_*n*K_CACHE_SIZE corresponding to the required block size *n* has been set appropriately.

See Also:

- *Oracle Database Administrator's Guide*

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 NLS_SORT='FRENCH';
CREATE INDEX test_idx ON test4(NLSSORT(name, 'NLS_SORT=FRENCH'));
SELECT * FROM test4 ORDER BY col;
```

```
ALTER SESSION SET NLS_COMP=LINGUISTIC;
SELECT * FROM test4 WHERE name > 'Henri';
```

Note: The SQL functions `MAX( )` and `MIN( )` cannot use linguistic indexes when `NLS_COMP` is set to `LINGUISTIC`.

Searching Linguistic Strings

Searching and collation are related tasks. Organizing data and processing it in a linguistically meaningful order is necessary for proper business processing. Searching and matching data in a linguistically meaningful way depends on what collation order is applied. For example, searching for all strings greater than `c` and less than `f` produces different results depending on the value of `NLS_SORT`. In an ASCII binary collation, the search finds any strings that start with `d` or `e` but excludes entries that begin with upper case `D` or `E` or accented `e` with a diacritic, such as ê. Applying an accent-insensitive binary collation returns all strings that start with `d`, `D`, and accented `e`, such as Ê or ê. Applying the same search with `NLS_SORT` set to `XSPANISH` also returns strings that start with `ch`, because `ch` is treated as a composite character that collates between `c` and `d` in traditional Spanish. This chapter discusses the kinds of collation that Oracle Database offers and how they affect string searches by SQL and SQL regular expressions.

See Also:

- "Linguistic Collation Features" on page 5-7
- "SQL Regular Expressions in a Multilingual Environment" on page 5-27

SQL Regular Expressions in a Multilingual Environment

Regular expressions provide a powerful method of identifying patterns of strings within a body of text. Usage ranges from a simple search for a string such as `San Francisco` to the more complex task of extracting all URLs to finding all words whose every second character is a vowel. SQL and PL/SQL support regular expressions in Oracle Database.

Traditional regular expression engines were designed to address only English text. However, regular expression implementations can encompass a wide variety of languages with characteristics that are very different from western European text. The implementation of regular expressions in Oracle Database is based on the Unicode Regular Expression Guidelines. The `REGEXP` SQL functions work with all character sets that are supported as database character sets and national character sets. Moreover, Oracle Database enhances the matching capabilities of the POSIX regular expression constructs to handle the unique linguistic requirements of matching multilingual data.

Oracle Database enhancements of the linguistic-sensitive operators are described in the following sections:

- Character Range '[x-y]' in Regular Expressions
- Collation Element Delimiter '[. .]' in Regular Expressions
- Character Class '[: :]' in Regular Expressions
- Equivalence Class '[= =]' in Regular Expressions
- Examples: Regular Expressions

See Also:

- *Oracle Database Development Guide* for more information about regular expression syntax
- *Oracle Database SQL Language Reference* for more information about `REGEX` SQL functions

Character Range '[x-y]' in Regular Expressions

According to the POSIX standard, a range in a regular expression includes all collation elements between the start point and the end point of the range in the linguistic definition of the current locale. Therefore, ranges in regular expressions are meant to be linguistic ranges, not byte value ranges, because byte value ranges depend on the platform, and the end user should not be expected to know the ordering of the byte values of the characters. The semantics of the range expression must be independent of the character set. This implies that a range such as `[a-d]` may include all the letters between `a` and `d` plus all of those letters with diacritics, plus any special case collation element such as `ch` in Traditional Spanish that is sorted as one character.

Oracle Database interprets range expressions as specified by the `NLS_SORT` parameter to determine the collation elements covered by a given range. For example:

```
Expression:     [a-d]e
NLS_SORT:       BINARY
Does not match: cheremoya
NLS_SORT:       XSPANISH
Matches:        >>che<<remoya
```

Collation Element Delimiter '[. .]' in Regular Expressions

This construct is introduced by the POSIX standard to separate collating elements. A **collating element** is a unit of collation and is equal to one character in most cases. However, the collation sequence in some languages may define two or more characters as a collating element. The historical regular expression syntax does not allow the user to define ranges involving multicharacter collation elements. For example, there was no way to define a range from `a` to `ch` because `ch` was interpreted as two separate characters.

By using the collating element delimiter `[. .]`, you can separate a multicharacter collation element from other elements. For example, the range from `a` to `ch` can be written as `[a-[.ch.]]`. It can also be used to separate single-character collating elements. If you use `[. .]` to enclose a multicharacter sequence that is not a defined collating element, then it is considered as a semantic error in the regular expression. For example, `[.ab.]` is considered invalid if `ab` is not a defined multicharacter collating element.

Character Class '[: :]' in Regular Expressions

In English regular expressions, the range expression can be used to indicate a character class. For example, `[a-z]` can be used to indicate any lowercase letter. However, in non-English regular expressions, this approach is not accurate unless `a` is the first lowercase letter and `z` is the last lowercase letter in the collation sequence of the language.

The POSIX standard introduces a new syntactical element to enable specifying explicit character classes in a portable way. The `[: :]` syntax denotes the set of characters belonging to a certain character class. The character class definition is based on the character set classification data.

Equivalence Class '[= =]' in Regular Expressions

Oracle Database also supports equivalence classes through the `[= =]` syntax as recommended by the POSIX standard. A base letter and all of the accented versions of the base constitute an **equivalence class**. For example, the equivalence class `[=a=]` matches `ä` as well as `â`. The current implementation does not support matching of Unicode composed and decomposed forms for performance reasons. For example, ä (`a` umlaut) does not match 'a followed by umlaut'.

Examples: Regular Expressions

The following examples show regular expression matches.

Example 5–16 Case-Insensitive Match Using the NLS_SORT Value

Case sensitivity in an Oracle Database regular expression match is determined at two levels: the `NLS_SORT` initialization parameter and the run-time match option. The `REGEXP` functions inherit the case-sensitivity behavior from the value of `NLS_SORT` by default. The value can also be explicitly overridden by the run-time match option `'c'` (case sensitive) or `'i'` (case insensitive).

```
Expression: catalog(ue)?
NLS_SORT: GENERIC_M_CI
Matches:
>>Catalog<<
>>catalogue<<
>>CATALOG<<
```

Oracle Database SQL syntax:

```
SQL> ALTER SESSION SET NLS_SORT='GENERIC_M_CI';
SQL> SELECT col FROM test WHERE REGEXP_LIKE(col,'catalog(ue)?');
```

Example 5–17 Case Insensitivity Overridden by the Run-time Match Option

```
Expression: catalog(ue)?
NLS_SORT: GENERIC_M_CI
Match option: 'c'
Matches:
>>catalogue<<
Does not match:
Catalog
CATALOG
```

Oracle Database SQL syntax:

```
SQL> ALTER SESSION SET NLS_SORT='GENERIC_M_CI';
SQL> SELECT col FROM test WHERE REGEXP_LIKE(col,'catalog(ue)?','c');
```

Example 5–18 Matching with the Collation Element Operator [..]

```
Expression: [^-a-[.ch.]]+  /*with NLS_SORT set to xspanish*/
Matches:
>>driver<<
Does not match:
cab
```

Oracle Database SQL syntax:

```
SQL> SELECT col FROM test WHERE REGEXP_LIKE(col,'[^-a-[.ch.]]+');
```

Example 5–19 Matching with the Character Class Operator [::]

This expression looks for 6-character strings with lowercase characters. Note that accented characters are matched as lowercase characters.

```
Expression: [[:lower:]]{6}
Database character set: WE8ISO8859P1
Matches:
>>maître<<
>>mòbile<<
>>pájaro<<
>>zurück<<
```

Oracle Database SQL syntax:

```
SQL> SELECT col FROM test WHERE REGEXP_LIKE(col,'[[:lower:]]{6}');
```

Example 5–20 Matching with the Base Letter Operator [==]

```
Expression: r[[=e=]]sum[[=e=]]
Matches:
>>resume<<
>>résumé<<
>>résume<<
>>resumé<<
```

Oracle Database SQL syntax:

```
SQL> SELECT col FROM test WHERE REGEXP_LIKE(col,'r[[=e=]]sum[[=e=]]');
```

See Also:

- *Oracle Database Development Guide* for more information about regular expression syntax
- *Oracle Database SQL Language Reference* for more information about `REGEX` SQL functions

6

Supporting Multilingual Databases with Unicode

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

- What is the Unicode Standard?
- Features of the Unicode Standard
- Implementing a Unicode Solution in the Database
- Unicode Case Studies
- Designing Database Schemas to Support Multiple Languages

What is the Unicode Standard?

The Unicode Standard is a character encoding system that defines every character in most of the spoken languages in the world.

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

The Unicode Standard, which is now in wide use, meets all of the requirements and capabilities of a global character set. It provides a unique code value for every character, regardless of the platform, program, or language. It also defines a number of character properties and processing rules that help implement complex multilingual text processing correctly and consistently. Bi-directional behavior, word breaking, and line breaking are examples of such complex processing.

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

Oracle Database introduced the Unicode Standard character encoding as the now obsolete database character set AL24UTFFSS in Oracle Database 7. Since then , incremental improvements have been made in each release to synchronize the support with the new published version of the standard. Oracle Database 12*c* (12.1.0.2) expands the support to version 6.2 of the Unicode Standard and introduces new linguistic collations complying with the Unicode Collation Algorithm (UCA).

See Also:

- `http://www.unicode.org` for more information about the Unicode Standard
- Chapter 5, "Linguistic Sorting and Matching" for more information about the support for the Unicode Collation Algorithm

Features of the Unicode Standard

This section contains the following topics:

- Code Points and Supplementary Characters
- Unicode Encoding Forms
- Support for the Unicode Standard in Oracle Database

Code Points and Supplementary Characters

The first version of the Unicode Standard was a 16-bit, fixed-width encoding that used two bytes to encode each character. This enabled 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.

The current definition of the Unicode Standard assigns a number to each character defined in the standard. These numbers are called code points, and are in the range 0 to 10FFFF hexadecimal. The Unicode notation for representing character code points is the prefix "U+" followed by the hexadecimal code point value. The code point value is left-padded with non-significant zeros to the minimum length of four. Characters with code points U+0000 to U+FFFF are called Basic Multilingual Plane characters. Characters with code points U+10000 to U+10FFFF are called supplementary characters.

Adding supplementary characters has increased the complexity of the Unicode 16-bit, fixed-width encoding form; however, this is still far less complex than managing hundreds of legacy encodings used before Unicode.

Unicode Encoding Forms

The Unicode Standard defines a few encoding forms, which are mappings from Unicode code points to code units. Code units are integer values processed by applications. Code units may have 8, 16, or 32 bits. The standard encoding forms are: UTF-8, UTF-16, and UTF-32. There are also two compatibility encodings mentioned in the standard and its associated technical reports: UCS-2 and CESU-8. Conversion

between different Unicode encodings is a simple bit-wise operation that is defined in the standard.

This section contains the following topics:

UTF-8 Encoding Form

UTF-8 is the 8-bit encoding form 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 byte representation. One Unicode character can be represented by 1 byte, 2 bytes, 3 bytes, or 4 bytes in the UTF-8 encoding form. Characters from the European and Middle Eastern 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 used for HTML and most Internet browsers.

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 character sets and UTF-8

See Also:

- "Code Points and Supplementary Characters" on page 6-2
- Table B–2, " Unicode Character Code Ranges for UTF-8 Character Codes" on page B-2

UTF-16 Encoding Form

UTF-16 is the 16-bit encoding form of Unicode. One character can be represented by either one 16-bit integer value (two bytes) or two 16-bit integer values (four bytes) in UTF-16. All characters from the Basic Multilingual Plane, which are most characters used in everyday text, are represented in two bytes. Supplementary characters are represented in four bytes. The two code units (integer values) encoding a single supplementary character are called a surrogate pair.

UTF-16 is the main Unicode encoding used for internal processing by Java since version J2SE 5.0 and by Microsoft Windows since version 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:

- "Code Points and Supplementary Characters" on page 6-2
- Table B–1, " Unicode Character Code Ranges for UTF-16 Character Codes" on page B-1

UCS-2 Encoding Form

UCS-2 is not an official Unicode encoding form. The name originally comes from older versions of the ISO/IEC 10646 standard, before the introduction of the supplementary characters. Therefore, it is currently used to refer to the UTF-16 encoding form stripped from support for supplementary characters and surrogate pairs. That is, surrogate pairs are processed in UCS-2 as two separate characters. Applications supporting UCS-2 but not UTF-16 should not process text containing supplementary characters, as they may incorrectly split surrogate pairs when dividing text into fragments. They are also generally incapable of displaying such text.

UCS-2 is the Unicode encoding used for internal processing by Java before version J2SE 5.0 and by Microsoft Windows NT.

UTF-32 Encoding Form

UTF-32 is the 32-bit encoding form of Unicode. Each Unicode code point is represented by a single 32-bit, fixed-width integer value. If is the simplest encoding form, but very space inefficient. For English text, it quadruples the storage requirements compared to UTF-8 and doubles when compared to UTF16. Therefore, UTF-32 is sometimes used as an intermediate form in internal text processing, but it is generally not used for information interchange.

In Java, since version J2SE 5.0, selected APIs have been enhanced to operate on characters in the 32-bit form, stored as `int` values.

CESU-8 Encoding Form

CESU-8 is not part of the core Unicode Standard. It is described in the Unicode Technical Report #26. CESU-8 is a compatibility encoding form identical to UTF-8 except for its representation of supplementary characters. In CESU-8, supplementary characters are represented as surrogate pairs, as in UTF-16. To obtain the CESU-8 encoding of a supplementary character, encode the character in UTF-16 first and then treat each of the surrogate code units as a code point with the same value. Then, apply the UTF-8 encoding rules (bit transformation) to each of the code points. This will yield two three-byte representations, six bytes in total.

CESU-8 has only two benefits:

- It has the same binary sorting order as UTF-16.
- It uses the same number of codes per character (one or two). This is important for character length semantics in string processing.

In general, the CESU-8 encoding form should be avoided as much as possible.

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

Figure 6–1 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.

Figure 6–1 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

Support for the Unicode Standard in Oracle Database

Oracle Database began supporting the Unicode character set as a database character set in release 7. Table 6–1 summarizes the Unicode character sets supported by Oracle Database.

Table 6–1 Unicode Character Sets Supported by Oracle Database

Character Set	Supported in RDBMS Release	Unicode Encoding Form	Unicode Version	Database Character Set	National Character Set
AL24UTFFSS	7.2 - 8*i*	UTF-8	1.1	Yes	No
UTF8	8.0 - 12*c*	CESU-8	For Oracle Database release 8.0 through Oracle8*i* Release 8.1.6: 2.1 For Oracle8*i* Database release 8.1.7 and later: 3.0	Yes	Yes (Oracle9*i* Database and newer only)

Table 6–1 (Cont.) Unicode Character Sets Supported by Oracle Database

Character Set	Supported in RDBMS Release	Unicode Encoding Form	Unicode Version	Database Character Set	National Character Set
UTFE	8.0 - 12*c*	UTF-EBCDIC	For Oracle8*i* Database releases 8.0 through 8.1.6: 2.1	Yes (see Note 1)	No
			For Oracle8*i* Database release 8.1.7 and later: 3.0		
AL32UTF8	9*i* - 12*c*	UTF-8	Oracle9*i* Database release 1: 3.0	Yes	No
			Oracle9*i* Database release 2: 3.1		
			Oracle Database 10*g*, release 1: 3.2		
			Oracle Database 10*g*, release 2: 4.0		
			Oracle Database 11*g*: 5.0		
			Oracle Database 12*c*: 6.2		
AL16UTF16	9*i* - 12*c*	UTF-16	Oracle9*i* Database release 1: 3.0	No	Yes
			Oracle9*i* Database release 2: 3.1		
			Oracle Database 10*g*, release 1: 3.2		
			Oracle Database 10*g*, release 2: 4.0		
			Oracle Database 11*g*: 5.0		
			Oracle Database 12*c*: 6.2		

Note 1: UTF-EBCDIC is a compatibility encoding form specific to EBCDIC-based systems, such as IBM z/OS or Fujitsu BS2000. It is described in the Unicode Technical Report #16. Oracle character set UTFE is a partial implementation of the UTF-EBCDIC encoding form, supported on ECBDIC-based platforms only. Oracle Database does not support five-byte sequences of the this encoding form, limiting the supported code point range to U+000 - U+3FFFF. The use of the UTFE character set is discouraged.

Implementing a Unicode Solution in the Database

Unicode characters can be stored in an Oracle database in two ways:

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

Note: You can combine both Unicode solutions, if required by different applications running in a single database.

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

- Enabling Multilingual Support for Whole Databases
- Enabling Multilingual Support with Unicode Data Types
- How to Choose Between Unicode Solutions

Enabling Multilingual Support for Whole Databases

The database character set specifies the encoding to be used in the SQL `CHAR` data types as well as the metadata such as table names, column names, and SQL statements. A **Unicode Standard-enabled database** is a database with a Unicode Standard-compliant character set as the database character set. There are two database Oracle character sets that implement the Unicode Standard.

- AL32UTF8

  The AL32UTF8 character set implements the UTF-8 encoding form and supports the latest version of the Unicode standard. It encodes characters in one, two, three, or four bytes. Supplementary characters require four bytes. It is for ASCII-based platforms.

  AL32UTF8 is the recommended database character set for any new deployment of Oracle Database as it provides the optimal support for multilingual applications, such as Internet websites and applications for multinational companies.

- UTF8

  The UTF8 character set implements the CESU-8 encoding form and encodes characters in one, two, or three bytes. It is for ASCII-based platforms.

  Supplementary characters inserted into a UTF8 database are stored in the CESU-8 encoding form. Each character is represented by two three-byte codes and hence occupies six bytes of memory in total.

  The properties of characters in the UTF8 character set are not guaranteed to be updated beyond version 3.0 of the Unicode Standard.

  Oracle recommends that you switch to AL32UTF8 for full support of the supplementary characters and the most recent versions of the Unicode Standard.

Example 6–1 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 = '+00:00';
```

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

Enabling Multilingual Support with Unicode Data Types

An alternative to storing Unicode data in the database is to use the SQL NCHAR data types (NCHAR, NVARCHAR2, NCLOB). You can store Unicode characters in columns of these data types regardless of how the database character set has been defined. The NCHAR data type is exclusively a Unicode data type, which means that it stores data encoded in a Unicode encoding form.

Oracle recommends using SQL CHAR, VARCHAR2, and CLOB data types in AL32UTF8 database to store Unicode character data. SQL NCHAR, NVARCHAR2, and NCLOB data types are not supported by some database features. Most notably, Oracle Text and XML DB do not support these data types.

You can create a table using the NVARCHAR2 and NCHAR data types. The column length specified for the NCHAR and NVARCHAR2 columns always equals 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 data types 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 and recommended for SQL NCHAR data types. This character set encodes Unicode data in the UTF-16 encoding form. It supports supplementary characters, which are stored as four bytes.

- UTF8

  When UTF8 is specified for SQL NCHAR data types, the data stored in the SQL data types is in CESU-8 encoding form. The UTF8 character set is deprecated.

You can specify the national character set for the SQL NCHAR data types 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 6–2 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 = '+00:00';
```

How to Choose Between Unicode Solutions

Oracle recommends that you deploy all new Oracle databases in the database character set AL32UTF8 and you use SQL `VARCHAR2`, `CHAR`, and `CLOB` data types to store character data. The SQL `NVARCHAR2`, `NCHAR`, and `NCLOB` data types should be considered only if:

- You have an existing database with a non-Unicode database character set and a legacy application, for which the business costs of migrating to Unicode would be inacceptable, and you need to add support for multilingual data in a small part of the application or in a small new module for which a separate database would not make much sense, or
- You need to create an application that has to support multilingual data and which must be installable in any of Oracle database deployed by your customers.

For the database character set in a Unicode Standard-enabled database, always select AL32UTF8. For the national character set, select AL16UTF16. If you consider choosing the deprecated UTF8 because of the lower storage requirements for English character data, first consider other options, such as data compression or increasing disk storage. Later migration to AL16UTF16 may be expensive, if a lot of data accumulates in the database.

Unicode Case Studies

This section describes typical scenarios for storing Unicode characters in an Oracle database:

- Example 6–3, "Unicode Solution with a Unicode Standard-Enabled Database"
- Example 6–4, "Unicode Solution with Unicode Data Types"

Example 6–3 Unicode Solution with a Unicode Standard-Enabled 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 about 300 GB, with very little data stored in `CLOB` columns.
- There is a nightly downtime of 4 hours.

In this case, a typical solution is to choose AL32UTF8 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 a Unicode character set is vital. Because the database is in US7ASCII, the easiest and fastest way of enabling the database to support the Unicode Standard is to switch the database character set to AL32UTF8 by using the Database Migration Assistant for Unicode (DMU). No data conversion is required for columns other than `CLOB` because US7ASCII is a subset of AL32UTF8.
- Because most of the code is written in Java and PL/SQL, changing the database character set to AL32UTF8 is unlikely to break existing code. Unicode support is automatically enabled in the application.

Example 6–4 Unicode Solution with Unicode Data Types

A European company that runs its legacy applications mainly on Windows platforms wants to add a new small Windows application written in Visual C/C++. The new application 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 WE8MSWIN1252.
- All character data in the existing database is composed of Western European characters.
- The database is around 500 GB with a lot of `CLOB` columns.
- Support for full-text search and XML storage is not required in the new application

A typical solution is to take the following actions:

- Use `NCHAR` and `NVARCHAR2` data types to store Unicode characters
- Keep WE8MSWIN1252 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 requires data conversion because the database character set is WE8MSWIN1252 (a Windows Latin-1 character set), which is not a subset of AL32UTF8. Also, a lot of data is stored in `CLOB` columns. All `CLOB` values in a database, even if they contain only ASCII characters, must be converted when migrating from a single-byte database character set, such as US7ASCII or WE8MSWIN1252 to AL32UTF8. As a result, there will be a lot of overhead in converting the data to AL32UTF8.
- The additional languages are supported in the new application only. It does not depend on the existing applications or schemas. It is simpler to use the Unicode data type in the new schema and keep the existing schemas unchanged.
- Only customer name columns require Unicode character set support. Using a single `NCHAR` column meets the customer's requirements without migrating the entire database.
- The new application does not need database features that do not support SQL `NCHAR` data types.
- The lengths of the SQL `NCHAR` data types are defined as number of characters. This is the same as how 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.

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 LOB Data Types
- Creating Indexes for Searching Multilingual Document Contents

Specifying Column Lengths for Multilingual Data

When you use NCHAR and NVARCHAR2 data types for storing multilingual data, the column size specified for a column is defined in number of characters. (This number of characters means the number of encoded Unicode code points, except that supplementary Unicode characters represented through surrogate pairs count as two characters.) Table 6–2 shows the maximum size of the NCHAR and NVARCHAR2 data types for the AL16UTF16 and UTF8 national character sets.

Table 6–2 Maximum Data Type Size

National Character Set	Maximum Column Size of NCHAR Data Type	Maximum Column Size of NVARCHAR2 Data Type (When MAX_STRING_SIZE = STANDARD)	Maximum Column Size of NVARCHAR2 Data Type (When MAX_STRING_SIZE = EXTENDED)
AL16UTF16	1000 characters	2000 characters	16383 characters
UTF8	2000 characters	4000 characters	32767 characters

This maximum size in characters is a constraint, not guaranteed capacity of the data type. The maximum capacity is expressed in bytes. For the NCHAR data type, the maximum capacity is 2000 bytes. For NVARCHAR2, it is 4000 bytes, if the initialization parameter MAX_STRING_SIZE is set to STANDARD, and 32767 bytes, if the initialization parameter MAX_STRING_SIZE is set to EXTENDED. When the national character set is AL16UTF16, the maximum number of characters never occupies more bytes than the maximum capacity, as each character (in an Oracle sense) occupies exactly 2 bytes. However, if the national character set is UTF8, the maximum number of characters can be stored only if all these characters are from the Unicode Basic Latin range, which corresponds to the ASCII standard. Other Unicode characters occupy more than one byte each in UTF8 and presence of such characters in a 4000 character string makes the string longer than the maximum 4000 bytes. If you want national character set columns to be able to hold the declared number of characters in any national character set, do not declare NCHAR columns longer than 2000/3=666 characters and NVARCHAR2 columns longer than 4000/3=1333 or 32767/3=10922 characters, depending on the MAX_STRING_SIZE initialization parameter.

When you use CHAR and VARCHAR2 data types 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. Application designers should consider using an extended character data type or CLOB data type if they need to store data larger than 4000 bytes.

See Also:

- *Oracle Database SQL Language Reference*
- *Oracle Database Reference* for more information about extending character data types by setting `MAX_STRING_SIZE` to the value of `EXTENDED`

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 Swedish or German character. In order to provide information in the language a user desires, data stored in a Unicode database should be tagged with 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 discuss example steps to achieve this goal:

- 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` data type 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, Oracle 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 a 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:** *Oracle Database Development Guide* for more information about fine-grained access control

Storing Documents in Multiple Languages in LOB Data Types

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

Data in CLOB columns is stored in the AL16UTF16 character set when the database character set is multibyte, such as UTF8 or AL32UTF8. 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 AL32UTF8, typically around 30% less, depending on the contents of the document.

Documents in NCLOB format are also stored in the AL16UTF16 character set regardless of the database character set or national character set. The storage space requirement is the same as for CLOB data. 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 with Oracle Text is not 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 data type.

Table 6–3 lists the advantages and disadvantages of the CLOB, NCLOB, and BLOB data types when storing documents:

Table 6–3 Comparison of LOB Data Types for Document Storage

Data Types	Advantages	Disadvantages
CLOB	■ Content search support with Oracle Text ■ 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 in CLOB format and BLOB format. 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 describes the high level steps to create indexes for documents in multiple languages. It contains the following topics:

- Creating Multilexers
- Creating Indexes for Documents Stored in the CLOB Data Type
- Creating Indexes for Documents Stored in the BLOB Data Type

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 in the CLOB Data Type

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 the `CLOB` data type 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 in the BLOB Data Type

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.

7

Programming with Unicode

This chapter describes how to use programming and access products for Oracle Database with Unicode. This chapter contains the following topics:

Overview of Programming with Unicode

Oracle 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 data type, such as `NCHAR` or `CHAR`, used in the database.

To avoid overloading the database server with data conversion operations, Oracle 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

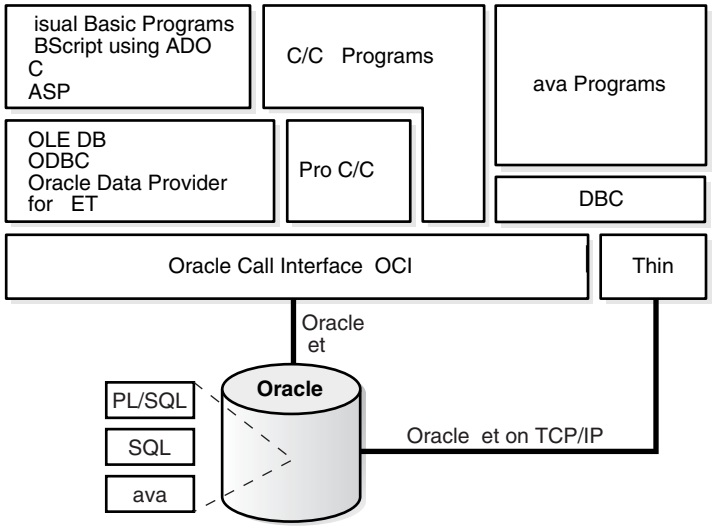
Oracle offers a comprehensive set of database access products that enable programs from different development environments to access Unicode data stored in the database. These products are listed in Table 7–1.

Table 7–1 Oracle Database Access Products

Programming Environment	Oracle Database Access Products
C/C++	Oracle Call Interface (OCI)
	Oracle Pro*C/C++
	Oracle ODBC driver
	Oracle Provider for OLE DB
	Oracle Data Provider for .NET
Java	Oracle JDBC OCI or thin driver
	Oracle server-side thin driver
	Oracle server-side internal driver
PL/SQL	Oracle PL/SQL and SQL
Visual Basic/C#	Oracle ODBC driver
	Oracle Provider for OLE DB

Figure 7–1 shows how the database access products can access the database.

Figure 7–1 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` data types. 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` data types.

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` data types of the database. It provides UTF-16 data access by implementing the `SQLWCHAR` interface specified in the ODBC standard specification.

The Oracle Provider for OLE DB enables C/C++, Visual Basic, and VBScript programs running on Windows platforms to access Unicode data stored in SQL `CHAR` and `NCHAR` data types. It provides UTF-16 data access through wide string OLE DB data types.

The Oracle Data Provider for .NET enables programs running in any .NET programming environment on Windows platforms to access Unicode data stored in SQL `CHAR` and `NCHAR` data types. It provides UTF-16 data access through Unicode data types.

Oracle JDBC drivers are the primary Java programmatic interface for accessing an Oracle database. Oracle provides the following 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
- The JDBC server-side thin driver, a pure Java driver used inside Java stored procedures to connect to another Oracle server
- The JDBC server-side internal driver that is used inside the Oracle server to access the data in the database

All drivers support Unicode data access to SQL `CHAR` and `NCHAR` data types 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 `CHAR`, `VARCHAR2`, `NCHAR`, and `NVARCHAR2` variables and to access SQL `CHAR` and `NCHAR` data types in the database.

The following sections describe how each of the database access products supports Unicode data access to an Oracle 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 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 Data Types
- Implicit Data Type Conversion Between NCHAR and Other Data Types
- Exception Handling for Data Loss During Data Type Conversion
- Rules for Implicit Data Type Conversion

SQL NCHAR Data Types

There are three SQL NCHAR data types:

The NCHAR Data Type

When you define a table column or a PL/SQL variable as the NCHAR data type, the length is always specified as the number of characters. For example, the following statement creates a column with a maximum length of 30 characters:

```
CREATE TABLE table1 (column1 NCHAR(30));
```

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 for each character)
```

For example, if the national character set is UTF8, then the maximum byte length is 30 characters times 3 bytes for each character, or 90 bytes.

The national character set, which is used for all NCHAR data types, is defined when the database is created. The national character set can be either UTF8 or AL16UTF16. The default is AL16UTF16.

The maximum column size allowed is 32000 characters when the national character set is UTF8 and 8000 when it is AL16UTF16. The actual data is subject to the maximum byte limit of 16000. 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, then 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 NVARCHAR2 data type or changing to the AL16UTF16 character set for the NCHAR data type.

The NVARCHAR2 Data Type

The NVARCHAR2 data type 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 length for the NVARCHAR2 type is 4000 characters if MAX_STRING_SIZE = STANDARD or 32767 characters if MAX_STRING_SIZE = EXTENDED. These lengths are based on using UTF8; the values are 2000 and 16383 characters when using AL16UTF16.

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 statement creates a table with one NVARCHAR2 column whose maximum length in characters is 2000 and maximum length in bytes is 4000.

```
CREATE TABLE table2 (column2 NVARCHAR2(2000));
```

The NCLOB Data Type

NCLOB is a character large object containing Unicode characters, with a maximum size of 4 gigabytes. Unlike the BLOB data type, the NCLOB data type has 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 in a format that is compatible with UCS-2, 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 a format that is compatible with UCS-2 before storing it in the database.

> **See Also:** *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about the NCLOB data type

Implicit Data Type Conversion Between NCHAR and Other Data Types

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

Type conversions between SQL CHAR data types and SQL NCHAR data types 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.

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

Exception Handling for Data Loss During Data Type Conversion

Data loss can occur during data type conversion when character set conversion is necessary. If a character in the source character set is not defined in the target character set, then a replacement character is 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, then the character is 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 stopped. 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 FALSE. This parameter works for both implicit and explicit conversion.

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

Rules for Implicit Data Type Conversion

In some cases, conversion between data types is possible in only one direction. In other cases, conversion in both directions is possible. Oracle defines a set of rules for conversion between data types. Table 7–2 contains the rules for conversion between data types.

Table 7–2 Rules for Conversion Between Data Types

Statement	Rule
INSERT/UPDATE statement	Values are converted to the data type of the target database column.
SELECT INTO statement	Data from the database is converted to the data type of the target variable.
Variable assignments	Values on the right of the equal sign are converted to the data type 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 data type is compared to a formal parameter of any of the CHAR, VARCHAR2, NCHAR or NVARCHAR2 data types. If the argument and formal parameter data types 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 data type and the other operand is a NUMBER or other non-character data type, then the other data type is converted to VARCHAR2 or NVARCHAR2. For concatenation between character data types, see "SQL NCHAR data types and SQL CHAR data types" on page 7-6.
SQL CHAR or NCHAR data types and NUMBER data type	Character values are converted to NUMBER data type.
SQL CHAR or NCHAR data types and DATE data type	Character values are converted to DATE data type.
SQL CHAR or NCHAR data types and ROWID data type	Character values are converted to ROWID data type.
SQL NCHAR data types and SQL CHAR data types	Comparisons between SQL NCHAR data types and SQL CHAR data types 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 data types and SQL CHAR data types, character set conversion occurs if they are encoded in different character sets. The character set for SQL NCHAR data types 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 data types 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 data types can always be converted to SQL NCHAR data types without data loss.

SQL Functions for Unicode Data Types

SQL NCHAR data types can be converted to and from SQL CHAR data types and other data types 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 7–1 Populating the Customers 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 7–2 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 7–3 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 data types can mixed because they are converted at run time.

```
DECLARE
ndatestring NVARCHAR2(20) := N'12-SEP-1975';
ndstr NVARCHAR2(50);
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 7–3, 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:** *Oracle Database SQL Language Reference* for more information about explicit conversion functions for SQL NCHAR data types

Other SQL Functions

Most SQL functions can take arguments of SQL NCHAR data types as well as mixed character data types. The return data type is based on the type of the first argument. If a non-string data type like NUMBER or DATE is passed to these functions, then it is converted to VARCHAR2. The following examples use the customer table created in "SQL Functions for Unicode Data Types" on page 7-7.

Example 7–4 INSTR Function

In this example, the string literal `'Sm'` is converted to `NVARCHAR2` and then scanned by `INSTR`, to detect the position of the first occurrence of this string in `name`.

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

Example 7–5 CONCAT Function

```
SELECT CONCAT(name,id) FROM customers;
```

`id` is converted to `NVARCHAR2` and then concatenated with `name`.

Example 7–6 RPAD Function

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

The following output results:

```
RPAD(NAME,100,'')
----------------------------------------
John Smith
```

The 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: *Oracle Database SQL Language Reference*

Unicode String Literals

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

- Put a prefix `N` before a string literal that is enclosed with single quotation marks. This explicitly indicates that the following string literal is an `NCHAR` string literal. For example, `N'résumé'` is an `NCHAR` string literal. For information about limitations of this method, see "NCHAR String Literal Replacement" on page 7-9.
- Use the `NCHR(n)` SQL function, which returns a unit of character code in the national character set, which is AL16UTF16 or UTF8. 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 with the same 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')` converts a string to the national character set. To ensure portability and to preserve data, include only ASCII characters and Unicode encoding in the following form: `\xxxx`, where `xxxx` is the hexadecimal value of a character code value in UTF-16 encoding format. For example, `UNISTR('G\0061ry')` represents `'Gary'`. The ASCII characters are converted to the database character set and then to the national character set. The Unicode encoding is converted directly to the national character set.

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

NCHAR String Literal Replacement

This section provides information on how to avoid data loss when performing NCHAR string literal replacement.

Being part of a SQL or PL/SQL statement, the text of any literal, with or without the prefix N, is encoded in the same character set as the rest of the statement. On the client side, the statement is in the client character set, which is determined by the client character set defined in NLS_LANG, or specified in the OCIEnvNlsCreate() call, or predefined as UTF-16 in JDBC. On the server side, the statement is in the database character set.

- When the SQL or PL/SQL statement is transferred from client to the database server, its character set is converted accordingly. It is important to note that if the database character set does not contain all characters used in the text literals, then the data is lost in this conversion. This problem affects NCHAR string literals more than the CHAR text literals. This is because the N' literals are designed to be independent of the database character set, and should be able to provide any data that the client character set supports.

  To avoid data loss in conversion to an incompatible database character set, you can activate the NCHAR literal replacement functionality. The functionality transparently replaces the N' literals on the client side with an internal format. The database server then decodes this to Unicode when the statement is executed.

- The sections "Handling SQL NCHAR String Literals in OCI" on page 7-16 and "Using SQL NCHAR String Literals in JDBC" on page 7-23 show how to switch on the replacement functionality in OCI and JDBC, respectively. Because many applications, for example, SQL*Plus, use OCI to connect to a database, and they do not control NCHAR literal replacement explicitly, you can set the client environment variable ORA_NCHAR_LITERAL_REPLACE to TRUE to control the functionality for them. By default, the functionality is switched off to maintain backward compatibility.

Using the UTL_FILE Package with NCHAR Data

The UTL_FILE package handles Unicode national character set data of the NVARCHAR2 data type. NCHAR and NCLOB are supported through implicit conversion. The functions and procedures include the following:

- FOPEN_NCHAR

  This function opens a file in national character set mode for input or output, with the maximum line size specified. Even though the contents of an NVARCHAR2 buffer may be AL16UTF16 or UTF8 (depending on the national character set of the database), the contents of the file are always read and written in UTF8. See "Support for the Unicode Standard in Oracle Database" on page 6-5 for more information. UTL_FILE converts between UTF8 and AL16UTF16 as necessary.

- 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. The file must be opened in national character set mode, and must be encoded in the UTF8 character set. The expected buffer data type is NVARCHAR2. If a variable of another data type, such as NCHAR, NCLOB, or VARCHAR2 is specified, PL/SQL performs standard implicit conversion from NVARCHAR2 after the text is read.

- PUT_NCHAR

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle. The file must be opened in the national character set mode. The text string will be written in the UTF8 character set. The expected buffer data type is NVARCHAR2. If a variable of another data type is specified, PL/SQL performs implicit conversion to NVARCHAR2 before writing the text.

- PUT_LINE_NCHAR

  This procedure is equivalent to PUT_NCHAR, except that the line separator is appended to the written text.

- PUTF_NCHAR

  This procedure is a formatted version of a PUT_NCHAR procedure. It accepts a format string with formatting elements \n and %s, and up to five arguments to be substituted for consecutive instances of %s in the format string. The expected data type of the format string and the arguments is NVARCHAR2. If variables of another data type are specified, PL/SQL performs implicit conversion to NVARCHAR2 before formatting the text. Formatted text is written in the UTF8 character set to the file identified by the file handle. The file must be opened in the national character set mode.

The above functions and procedures process text files encoded in the UTF8 character set, that is, in the Unicode CESU-8 encoding. See "Universal Character Sets" on page A-16 for more information about CESU-8. The functions and procedures convert between UTF8 and the national character set of the database, which can be UTF8 or AL16UTF16, as needed.

> **See Also:** *Oracle Database 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
- Setting UTF-8 to the NLS_LANG Character Set in OCI
- Binding and Defining SQL CHAR Data Types in OCI
- Binding and Defining SQL NCHAR Data Types in OCI
- Binding and Defining CLOB and NCLOB Unicode Data in OCI

> **See Also:** Chapter 10, "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 data types (`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` data types 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 also must be 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: "Specifying Character Sets in OCI" on page 10-2

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, then 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 data types, 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 data types 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 data types 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 data type of the target columns. If you do not know the target data type, then you should set the OCI_ATTR_CHARSET_FORM attribute to SQLCS_NCHAR when binding and defining.

Table 7–3 contains information about OCI character set conversions.

Table 7–3 OCI Character Set Conversions

Data Types for OCI Client Buffer	OCI_ATTR_CHARSET_FORM	Data Types 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

Table 7–3 (Cont.) OCI Character Set Conversions

Data Types for OCI Client Buffer	OCI_ATTR_CHARSET_FORM	Data Types of the Target Column in the Database	Conversion Between	Comments
text	SQLCS_IMPLICIT	CHAR,VARCHAR2,CLOB	NLS_LANG character set and database character set in OCI	No unexpected data loss
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 must 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, then you must consider changing the table schema. For defining operations, client applications must 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 for each 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);
...
```

Setting UTF-8 to the NLS_LANG Character Set in OCI

For OCI client applications that support Unicode UTF-8 encoding, use AL32UTF8 to specify the `NLS_LANG` character set, unless the database character set is UTF8. Use UTF8 if the database character set is UTF8.

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 Data Types in OCI

To specify a Unicode character set for binding and defining data with SQL `CHAR` data types, 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` data type, then 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 Data Types in OCI

Oracle recommends that you access SQL NCHAR data types using UTF-16 binding or defining when using OCI. Beginning with Oracle9*i*, SQL NCHAR data types are Unicode data types with an encoding of either UTF8 or AL16UTF16. To access data in SQL NCHAR data types, 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 data types is always in the number of Unicode code units.

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);
...
```

Handling SQL NCHAR String Literals in OCI

By default, the NCHAR literal replacement is not performed in OCI. (Refer to "NCHAR String Literal Replacement" on page 7-9.)

You can switch it on by setting the environment variable ORA_NCHAR_LITERAL_REPLACE to TRUE. You can also achieve this behavior programmatically by using the OCI_NCHAR_LITERAL_REPLACE_ON and OCI_NCHAR_LITERAL_REPLACE_OFF modes in OCIEnvCreate() and OCIEnvNlsCreate(). So, for example, OCIEnvCreate(OCI_NCHAR_LITERAL_REPLACE_ON) turns on NCHAR literal replacement, while OCIEnvCreate(OCI_NCHAR_LITERAL_REPLACE_OFF) turns it off.

As an example, consider the following statement:

```
int main(argc, argv)
{
   OCIEnv *envhp;
if (OCIEnvCreate((OCIEnv **) &envhp,
   (ub4)OCI_THREADED|OCI_NCHAR_LITERAL_REPLACE_ON,
   (dvoid *)0, (dvoid * (*)(dvoid *, size_t)) 0,
   (dvoid * (*)(dvoid *, dvoid *, size_t))0,
   (void (*)(dvoid *, dvoid *)) 0,
   (size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIEnvCreate()\n";
  return 1;
}
...
}
```

Note that, when the NCHAR literal replacement is turned on, OCIStmtPrepare and OCIStmtPrepare2 transforms N' literals with U' literals in the SQL text and store the resulting SQL text in the statement handle. Thus, if the application uses OCI_ATTR_STATEMENT to retrieve the SQL text from the OCI statement handle, the SQL text returns U' instead of N' as specified in the original text.

> **See Also:** *Oracle Database Administrator's Guide* for information regarding environment variables

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 units. 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 units. Note one Unicode supplementary character is counted as two code units, because the encoding is UTF-16. After binding or defining a 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 units if you bind or define as UTF-16.

```
err = OCILobGetLength(ctx->svchp, ctx->errhp, lobp, &lenp);
```

If you are using an NCLOB, then 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++ data type or the native C/C++ text data type, a program can access Unicode data stored in SQL CHAR data types of a UTF8 or AL32UTF8 database. Alternatively, a program could use the C/C++ native text type.
- Using the UVARCHAR Pro*C/C++ data type or the native C/C++ utext data type, a program can access Unicode data stored in NCHAR data types of a database.
- Using the NVARCHAR Pro*C/C++ data type, a program can access Unicode data stored in NCHAR data types. The difference between UVARCHAR and NVARCHAR in a Pro*C/C++ program is that the data for the UVARCHAR data type is stored in a utext buffer while the data for the NVARCHAR data type is stored in a text data type.

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

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 data types used in a Pro*C/C++ program. Table 7–4 illustrates the conversion paths:

*Table 7–4 Pro*C/C++ Bind and Define Data Conversion*

Pro*C/C++ Data Type	SQL Data Type	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 Data Type in Pro*C/C++

The Pro*C/C++ VARCHAR data type is preprocessed to a struct with a length field and text buffer field. The following example uses the C/C++ text native data type and the VARCHAR Pro*C/C++ data types 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++ varchar 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 data type or native text data type in a Pro*C/C++ program, the preprocessor assumes that the program intends to access columns of SQL CHAR data types instead of SQL NCHAR data types 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 data types in the database, then implicit conversion occurs 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 Data Type in Pro*C/C++

The Pro*C/C++ NVARCHAR data type is similar to the Pro*C/C++ VARCHAR data type. It should be used to access SQL NCHAR data types in the database. It tells Pro*C/C++ preprocessor to bind or define a text buffer to the column of SQL NCHAR data types. The preprocessor specifies 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 data types, then 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 Data Type in Pro*C/C++

The UVARCHAR data type 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 data type or native utext data type in Pro*C/C++ programs, the preprocessor assumes that the program intends to access SQL NCHAR data types. 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 data type, then 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 Programming with Unicode

Oracle provides the following JDBC drivers for Java programs to access character data in an Oracle database:

- The JDBC OCI driver
- The JDBC thin driver

- The JDBC server-side internal driver
- The JDBC server-side thin driver

Java programs can insert or retrieve character data to and from columns of SQL CHAR and NCHAR data types. Specifically, JDBC enables Java programs to bind or define Java strings to SQL CHAR and NCHAR data types. Because Java's string data type 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. JDBC also enables you to specify the PL/SQL and SQL statements in Java strings so that any non-ASCII schema object names and string literals can be used.

At database connection time, JDBC 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 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 contains the following topics:

Binding and Defining Java Strings to SQL CHAR Data Types

Oracle JDBC drivers allow you to access SQL CHAR data types in the database using Java string bind or define variables. The following code illustrates how to bind a Java string to a CHAR column.

```
int employee_id = 12345;
String last_name = "Joe";
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO" +
    "employees (last_name, employee_id) VALUES (?, ?)");
pstmt.setString(1, last_name);
pstmt.setInt(2, employee_id);
pstmt.execute();                /* execute to insert into first row */
employee_id += 1;                     /* next employee number */
last_name = "\uFF2A\uFF4F\uFF45";    /* Unicode characters in name */
pstmt.setString(1, last_name);
pstmt.setInt(2, employee_id);
pstmt.execute();                /* execute to insert into second row */
```

You can define the target SQL columns by specifying their data types and lengths. When you define a SQL CHAR column with the data type and the length, JDBC uses this information to optimize the performance of fetching SQL CHAR data from the column. The following is an example of defining a SQL CHAR column.

```
OraclePreparedStatement pstmt = (OraclePreparedStatement)
     conn.prepareStatement("SELECT ename, empno from emp");
pstmt.defineColumnType(1,Types.VARCHAR, 3);
pstmt.defineColumnType(2,Types.INTEGER);
ResultSet rest = pstmt.executeQuery();
```

```
String name = rset.getString(1);
int id = reset.getInt(2);
```

You must cast PreparedStatement to OraclePreparedStatement to call defineColumnType(). The second parameter of defineColumnType() is the data type of the target SQL column. The third parameter is the length in number of characters.

Binding and Defining Java Strings to SQL NCHAR Data Types

For binding or defining Java string variables to SQL NCHAR data types, Oracle provides an extended PreparedStatement which has the setFormOfUse() method through which you can explicitly specify the target column of a bind variable to be a SQL NCHAR data type. The following code illustrates how to bind a Java string to an NCHAR column.

```
int employee_id = 12345;
String last_name = "Joe"
oracle.jdbc.OraclePreparedStatement pstmt =
    (oracle.jdbc.OraclePreparedStatement)
    conn.prepareStatement("INSERT INTO employees (last_name, employee_id)
    VALUES    (?, ?)");
pstmt.setFormOfUse(1, oracle.jdbc.OraclePreparedStatement.FORM_NCHAR);
pstmt.setString(1, last_name);
pstmt.setInt(2, employee_id);
pstmt.execute();                 /* execute to insert into first row */
employee_id += 1;                      /* next employee number */
last_name = "\uFF2A\uFF4F\uFF45";    /* Unicode characters in name */
pstmt.setString(1, last_name);
pstmt.setInt(2, employee_id);
pstmt.execute();                 /* execute to insert into second row */
```

You can define the target SQL NCHAR columns by specifying their data types, forms of use, and lengths. JDBC uses this information to optimize the performance of fetching SQL NCHAR data from these columns. The following is an example of defining a SQL NCHAR column.

```
OraclePreparedStatement pstmt = (OraclePreparedStatement)
     conn.prepareStatement("SELECT ename, empno from emp");
   pstmt.defineColumnType(1,Types.VARCHAR, 3,
OraclePreparedStatement.FORM_NCHAR);
   pstmt.defineColumnType(2,Types.INTEGER);
   ResultSet rest = pstmt.executeQuery();
   String name = rset.getString(1);
   int id = reset.getInt(2);
```

To define a SQL NCHAR column, you must specify the data type that is equivalent to a SQL CHAR column in the first argument, the length in number of characters in the second argument, and the form of use in the fourth argument of defineColumnType().

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, then 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 data types happens in the database, database performance is degraded.

In addition, if you bind or define a Java string for a column of SQL CHAR data types but specify the form of use argument, then performance of the database is degraded. However, data should not be lost because the national character set is always a larger set than the database character set.

New JDBC4.0 Methods for NCHAR Data Types

JDBC 11.1 adds support for the new JDBC 4.0 (JDK6) SQL data types NCHAR, NVARCHAR, LONGNVARCHAR, and NCLOB. To retrieve a national character value, an application can call one of the following methods:

- getNString
- getNClob
- getNCharacterStream

The getNClob method verifies that the retrieved value is indeed an NCLOB. Otherwise, these methods are equivalent to corresponding methods without the letter N.

To specify a value for a parameter marker of national character type, an application can call one of the following methods:

- setNString
- setNCharacterStream
- setNClob

These methods are equivalent to corresponding methods without the letter N preceded by a call to setFormOfUse(..., OraclePreparedStatement.FORM_NCHAR).

> **See Also:** *Oracle Database JDBC Developer's Guide* for more information

Using the SQL NCHAR Data Types Without Changing the Code

A Java system property has been introduced in the Oracle JDBC drivers for customers to tell whether the form of use argument should be specified by default in a Java application. This property has the following purposes:

- Existing applications accessing the SQL CHAR data types can be migrated to support the SQL NCHAR data types for worldwide deployment without changing a line of code.
- Applications do not need to call the setFormOfUse() method when binding and defining a SQL NCHAR column. The application code can be made neutral and independent of the data types being used in the back-end database. With this property set, applications can be easily switched from using SQL CHAR or SQL NCHAR.

The Java system property is specified in the command line that invokes the Java application. The syntax of specifying this flag is as follows:

```
java -Doracle.jdbc.defaultNChar=true <application class>
```

With this property specified, the Oracle JDBC drivers assume the presence of the form of use argument for all bind and define operations in the application.

If you have a database schema that consists of both the SQL CHAR and SQL NCHAR columns, then using this flag may have some performance impact when accessing the SQL CHAR columns because of implicit conversion done in the database server.

> **See Also:** "Data Conversion in JDBC" on page 7-23 for more information about the performance impact of implicit conversion

Using SQL NCHAR String Literals in JDBC

When using NCHAR string literals in JDBC, there is a potential for data loss because characters are converted to the database character set before processing. See "NCHAR String Literal Replacement" on page 7-9 for more details.

The desired behavior for preserving the NCHAR string literals can be achieved by enabling the property set oracle.jdbc.convertNcharLiterals. If the value is true, then this option is enabled; otherwise, it is disabled. The default setting is false. It can be enabled in two ways: a) as a Java system property or b) as a connection property. Once enabled, conversion is performed on all SQL in the VM (system property) or in the connection (connection property). For example, the property can be set as a Java system property as follows:

```
java -Doracle.jdbc.convertNcharLiterals="true" ...
```

Alternatively, you can set this as a connection property as follows:

```
Properties props = new Properties();
...
props.setProperty("oracle.jdbc.convertNcharLiterals", "true");
Connection conn = DriverManager.getConnection(url, props);
```

If you set this as a connection property, it overrides a system property setting.

Data Conversion in JDBC

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 JDBC drivers:

- Data Conversion for the OCI Driver
- Data Conversion for Thin Drivers
- Data Conversion for the Server-Side Internal 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. When the database character set is neither US7ASCII nor WE8ISO8859P1, the driver converts the SQL statements to UTF-8 first in Java and then to the database character set in C. Otherwise, it converts the SQL statements directly to the database character set. For Java string bind variables, Table 7–5 summarizes the conversion paths taken for different scenarios. For Java string define variables, the same conversion paths, but in the opposite direction, are taken.

Table 7–5 OCI Driver Conversion Path

Form of Use	SQL Data Type	Conversion Path
FORM_CHAR (Default)	CHAR	Conversion between the UTF-16 encoding of a Java string and the database character set happens in the JDBC driver.
FORM_CHAR (Default)	NCHAR	Conversion between the UTF-16 encoding of a Java string and the database character set happens in the JDBC driver. Then, conversion between the database character set and the national character set happens in the database server.
FORM_NCHAR	NCHAR	Conversion between the UTF-16 encoding of a Java string and the national character set happens in the JDBC driver.
FORM_NCHAR	CHAR	Conversion between the UTF-16 encoding of a Java string and the national character set happens in the JDBC driver. Then, conversion between the national character set and the database character set happens in the database server.

Data Conversion for Thin Drivers

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 driver converts the SQL statement to the database character set when the database character set is one of the following character sets:

- US7ASCII
- WE8ISO8859P1
- WE8DEC
- WE8MSWIN1252

Otherwise, the driver converts the SQL statement to UTF-8 and notifies the database that the statement requires further conversion before being processed. The database, in turn, converts the SQL statement to the database character set. For Java string bind variables, the conversion paths shown in Table 7–6 are taken for the thin driver. For Java string define variables, the same conversion paths but in the opposite direction are taken. The four character sets listed earlier are called **selected characters sets** in the table.

Table 7–6 Thin Driver Conversion Path

Form of Use	SQL Data Type	Database Character Set	Conversion Path
FORM_CHAR (Default)	CHAR	One of the selected character sets	Conversion between the UTF-16 encoding of a Java string and the database character set happens in the thin driver.
FORM_CHAR (Default)	NCHAR	One of the selected character sets	Conversion between the UTF-16 encoding of a Java string and the database character set happens in the thin driver. Then, conversion between the database character set and the national character set happens in the database server.
FORM_CHAR (Default)	CHAR	Other than the selected character sets	Conversion between the UTF-16 encoding of a Java string and UTF-8 happens in the thin driver. Then, conversion between UTF-8 and the database character set happens in the database server.

Table 7–6 (Cont.) Thin Driver Conversion Path

Form of Use	SQL Data Type	Database Character Set	Conversion Path
FORM_CHAR (Default)	NCHAR	Other than the selected character sets	Conversion between the UTF-16 encoding of a Java string and UTF-8 happens in the thin driver. Then, conversion from UTF-8 to the database character set and then to the national character set happens in the database server.
FORM_NCHAR	CHAR	Any	Conversion between the UTF-16 encoding of a Java string and the national character set happens in the thin driver. Then, conversion between the national character set and the database character set happens in the database server.
FORM_NCHAR	NCHAR	Any	Conversion between the UTF-16 encoding of a Java string and the national character set happens in the thin driver.

Data Conversion for the Server-Side Internal Driver

All data conversion occurs in the database server because the server-side internal driver works inside the database.

Using oracle.sql.CHAR in Oracle Object Types

JDBC drivers support Oracle object types. Oracle objects are always sent from database to client as an object represented in the database character set or national character set. That means the data conversion path in "Data Conversion in JDBC" on page 7-23 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.

This section includes the following topics:

- oracle.sql.CHAR
- Accessing SQL CHAR and NCHAR Attributes with oracle.sql.CHAR

oracle.sql.CHAR

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.

Accessing SQL CHAR and NCHAR Attributes with oracle.sql.CHAR

The following is an example of an object type created using SQL:

```
CREATE TYPE person_type AS OBJECT (
   name VARCHAR2(30), address NVARCHAR2(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` data type. 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_address = sql_address.getString();
```

The `getString()` method of the `oracle.sql.CHAR` class converts the byte array from the database character set or national 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 topic:

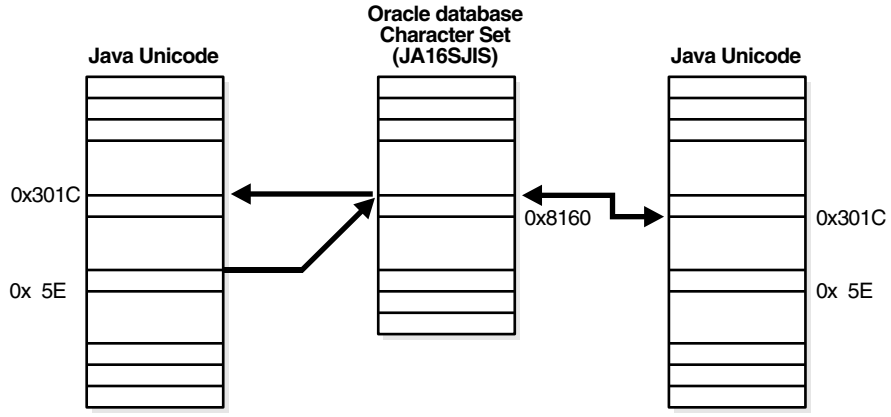
- Character Integrity Issues in a Multibyte Database Environment

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 7–2 shows the round-trip conversion of the tilde character.

Figure 7–2 Character Integrity



This issue is not a bug in Oracle's JDBC. It is an unfortunate side effect of the ambiguity in character mapping specifications 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.

ODBC and OLE DB Programming with Unicode

You should use the Oracle ODBC driver or Oracle Provider for OLE DB to access the Oracle server 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
- OCI Dependency in Unicode
- ODBC and OLE DB Code Conversion in Unicode
- ODBC Unicode Data Types
- OLE DB Unicode Data Types
- ADO Access

Unicode-Enabled Drivers in ODBC and OLE DB

Oracle's ODBC driver and Oracle Provider for OLE DB 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.

Oracle provides ODBC and OLE DB products for Windows platforms 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 7-10

ODBC and OLE DB Code Conversion in Unicode

In general, no redundant data conversion occurs unless you specify a different client data type from that of the server. If you bind Unicode buffer SQL_C_WCHAR with a Unicode data column like NCHAR, for example, then ODBC and OLE DB drivers bypass it between the application and OCI layer.

If you do not specify data types before fetching, but call SQLGetData with the client data types instead, then the conversions in Table 7–7 occur.

Table 7–7 ODBC Implicit Binding Code Conversions

Data Types of ODBC Client Buffer	Data Types 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 data type for inserting and updating operations.

The data type 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, then 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 7–8 OLE DB Implicit Bindings

Data Types of OLE_DB Client Buffer	Data Types 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 Data Types

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 data type identifiers for the SQL_C_WCHAR, SQL_C_WVARCHAR, and SQL_WLONGVARCHAR clients; and three Unicode data type identifiers for servers SQL_WCHAR, SQL_WVARCHAR, and SQL_WLONGVARCHAR.

For binding operations, specify data types 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 7–9 represents the data type mappings of the ODBC Unicode data types for the server against SQL NCHAR data types.

Table 7–9 Server ODBC Unicode Data Type Mapping

ODBC Data Type	Oracle Data Type
SQL_WCHAR	NCHAR

Table 7–9 (Cont.) Server ODBC Unicode Data Type Mapping

ODBC Data Type	Oracle Data Type
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 Data Types

OLE DB offers the wchar_t, BSTR, and OLESTR data types for a Unicode C client. In practice, wchar_t is the most common data type 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, then 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 data types. 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 data type explicitly. When you set the client data type, the OLE DB driver automatically performs data conversion if necessary.

Table 7–10 illustrates OLE DB data type mapping.

Table 7–10 OLE DB Data Type Mapping

OLE DB Data Type	Oracle Data Type
DBTYPE_WCHAR	NCHAR or NVARCHAR2

If DBTYPE_BSTR is specified, then 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.

XML Programming with Unicode

XML support of Unicode is essential for software development for global markets so that text information can be exchanged in any language. Unicode uniformly supports almost every character and language, which makes it much easier to support multiple

languages within XML. To enable Unicode for XML within an Oracle database, the character set of the database must be UTF-8. By enabling Unicode text handling in your application, you acquire a basis for supporting any language. Every XML document is Unicode text and potentially multilingual, unless it is guaranteed that only a known subset of Unicode characters will appear on your documents. Thus Oracle recommends that you enable Unicode for XML. Unicode support comes with Java and many other modern programming environments.

This section includes the following topics:

- Writing an XML File in Unicode with Java
- Reading an XML File in Unicode with Java
- Parsing an XML Stream in Unicode with Java

Writing an XML File in Unicode with Java

A common mistake in reading and writing XML files is using the `Reader` and `Writer` classes for character input and output. Using `Reader` and `Writer` for XML files should be avoided because it requires character set conversion based on the default character encoding of the run-time environment.

For example, using `FileWriter` class is not safe because it converts the document to the default character encoding. The output file can suffer from a parsing error or data loss if the document contains characters that are not available in the default character encoding.

UTF-8 is popular for XML documents, but UTF-8 is not usually the default file encoding for Java. Thus using a Java class that assumes the default file encoding can cause problems.

The following example shows how to avoid these problems:

```
import java.io.*;
import oracle.xml.parser.v2.*;

public class I18nSafeXMLFileWritingSample
{
  public static void main(String[] args) throws Exception
  {
    // create a test document
    XMLDocument         doc  = new XMLDocument();
    doc.setVersion( "1.0" );
    doc.appendChild(doc.createComment( "This is a test empty document." ));
    doc.appendChild(doc.createElement( "root" ));

    // create a file
    File                file = new File( "myfile.xml" );

    // create a binary output stream to write to the file just created
    FileOutputStream    fos  = new FileOutputStream( file );

    // create a Writer that converts Java character stream to UTF-8 stream
    OutputStreamWriter  osw  = new OutputStreamWriter( fos, "UTF8" );

    // buffering for efficiency
    Writer              w    = new BufferedWriter( osw );

    // create a PrintWriter to adapt to the printing method
    PrintWriter         out  = new PrintWriter( w );
```

```
    // print the document to the file through the connected objects
    doc.print( out );
  }
}
```

Reading an XML File in Unicode with Java

Do not read XML files as text input. When reading an XML document stored in a file system, use the parser to automatically detect the character encoding of the document. Avoid using a `Reader` class or specifying a character encoding on the input stream. Given a binary input stream with no external encoding information, the parser automatically figures out the character encoding based on the byte order mark and encoding declaration of the XML document. Any well-formed document in any supported encoding can be successfully parsed using the following sample code:

```
import java.io.*;
import oracle.xml.parser.v2.*;

public class I18nSafeXMLFileReadingSample
{
  public static void main(String[] args) throws Exception
  {
    // create an instance of the xml file
    File                file = new File( "myfile.xml" );

    // create a binary input stream
    FileInputStream     fis  = new FileInputStream( file );

    // buffering for efficiency
    BufferedInputStream  in = new BufferedInputStream( fis );

    // get an instance of the parser
    DOMParser  parser = new DOMParser();

    // parse the xml file
    parser.parse( in );
  }
}
```

Parsing an XML Stream in Unicode with Java

When the source of an XML document is not a file system, the encoding information is usually available before reading the document. For example, if the input document is provided in the form of a Java character stream or Reader, its encoding is evident and no detection should take place. The parser can begin parsing a Reader in Unicode without regard to the character encoding.

The following is an example of parsing a document with external encoding information:

```
import java.io.*;
import java.net.*;
import org.xml.sax.*;
import oracle.xml.parser.v2.*;

public class I18nSafeXMLStreamReadingSample
{
  public static void main(String[] args) throws Exception
  {
    // create an instance of the xml file
```

```
    URL  url = new URL( "http://myhost/mydocument.xml" );

    // create a connection to the xml document
    URLConnection  conn = url.openConnection();

    // get an input stream
    InputStream  is = conn.getInputStream();

    // buffering for efficiency
    BufferedInputStream  bis = new BufferedInputStream( is );

    /* figure out the character encoding here                              */
    /* a typical source of encoding information is the content-type header */
    /* we assume it is found to be utf-8 in this example                   */
    String  charset = "utf-8";

    // create an InputSource for UTF-8 stream
    InputSource  in = new InputSource( bis );
    in.setEncoding( charset );

    // get an instance of the parser
    DOMParser  parser = new DOMParser();

    // parse the xml stream
    parser.parse( in );
  }
}
```

8

Oracle Globalization Development Kit

This chapter includes the following sections:

Overview of the Oracle Globalization Development Kit

Designing and developing a globalized application can be a daunting task even for the most experienced developers. This is usually caused by lack of knowledge and the complexity of globalization concepts and APIs. Application developers who write applications using Oracle Database need to understand the Globalization Support architecture of the database, including the properties of the different character sets, territories, languages and linguistic sort definitions. They also need to understand the globalization functionality of their middle-tier programming environment, and find out how it can interact and synchronize with the locale model of the database. Finally, to develop a globalized Internet application, they need to design and write code that is capable of simultaneously supporting multiple clients running on different operating systems, with different character sets and locale requirements.

Oracle Globalization Development Kit (GDK) simplifies the development process and reduces the cost of developing Internet applications that will be used to support a global environment. The GDK includes comprehensive programming APIs for both Java and PL/SQL, code samples, and documentation that address many of the design, development, and deployment issues encountered while creating global applications.

The GDK mainly consists of two parts: GDK for Java and GDK for PL/SQL. GDK for Java provides globalization support to Java applications. GDK for PL/SQL provides globalization support to the PL/SQL programming environment. The features offered in GDK for Java and GDK for PL/SQL are not identical.

Designing a Global Internet Application

There are two architectural models for deploying a global Web site or a global Internet application, depending on your globalization and business requirements. Which model to deploy affects how the Internet application is developed and how the application server is configured in the middle-tier. The two models are:

- Multiple instances of monolingual Internet applications

  Internet applications that support only one locale in a single binary are classified as monolingual applications. A locale refers to a national language and the region in which the language is spoken. For example, the primary language of the United States and Great Britain is English. However, the two territories have different currencies and different conventions for date formats. Therefore, the United States and Great Britain are considered to be two different locales.

  This level of globalization support is suitable for customers who want to support one locale for each instance of the application. Users need to have different entry points to access the applications for different locales. This model is manageable only if the number of supported locales is small.

- Single instance of a multilingual application

  Internet applications that support multiple locales simultaneously in a single binary are classified as multilingual applications. This level of globalization support is suitable for customers who want to support several locales in an Internet application simultaneously. Users of different locale preferences use the same entry point to access the application.

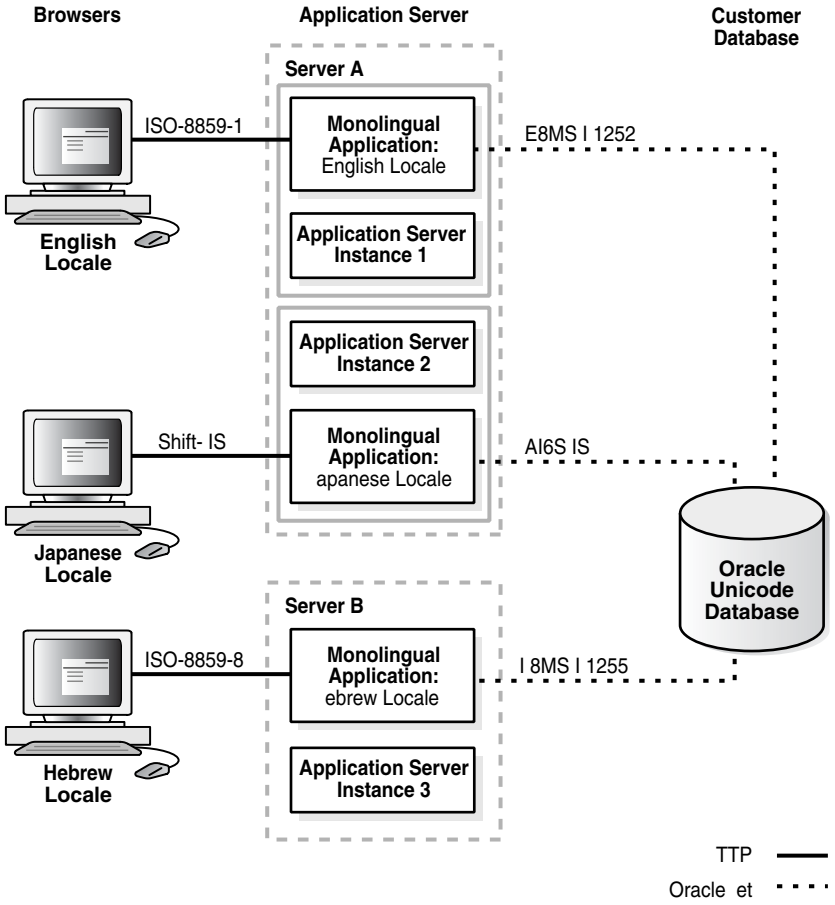
  Developing an application using the monolingual model is very different from developing an application using the multilingual model. The Globalization Development Kit consists of libraries, which can assist in the development of global applications using either architectural model.

The rest of this section includes the following topics:

Deploying a Monolingual Internet Application

Deploying a global Internet application with multiple instances of monolingual Internet applications is shown in Figure 8–1.

Figure 8–1 Monolingual Internet Application Architecture



Each application server is configured for the locale that it serves. This deployment model assumes that one instance of an Internet application runs in the same locale as the application in the middle tier.

The Internet applications access a back-end database in the native encoding used for the locale. The following are advantages of deploying monolingual Internet applications:

- The support of the individual locales is separated into different servers so that multiple locales can be supported independently in different locations and that the workload can be distributed accordingly. For example, customers may want to support Western European locales first and then support Asian locales such as Japanese (Japan) later.
- The complexity required to support multiple locales simultaneously is avoided. The amount of code to write is significantly less for a monolingual Internet application than for a multilingual Internet application.

The following are disadvantages of deploying monolingual Internet applications:

- Extra effort is required to maintain and manage multiple servers for different locales. Different configurations are required for different application servers.
- The minimum number of application servers required depends on the number of locales the application supports, regardless of whether the site traffic will reach the capacity provided by the application servers.

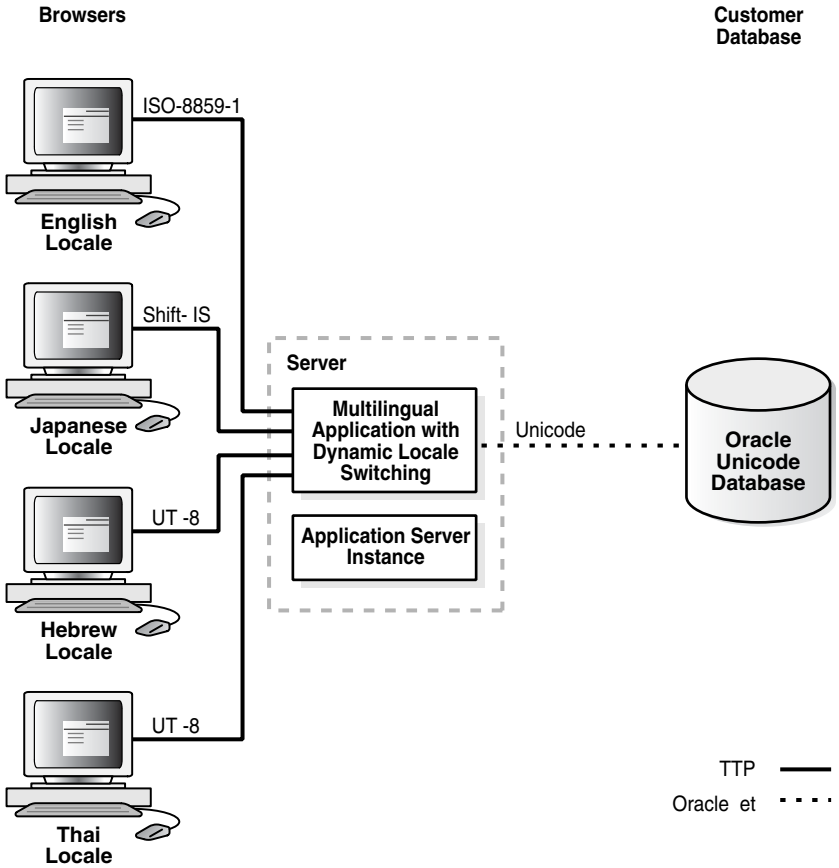
- Load balancing for application servers is limited to the group of application servers for the same locale.
- More QA resources, both human and machine, are required for multiple configurations of application servers. Internet applications running on different locales must be certified on the corresponding application server configuration.
- It is not designed to support multilingual content. For example, a web page containing Japanese and Arabic data cannot be easily supported in this model.

As more and more locales are supported, the disadvantages quickly outweigh the advantages. With the limitation and the maintenance overhead of the monolingual deployment model, this deployment architecture is suitable for applications that support only one or two locales.

Deploying a Multilingual Internet Application

Multilingual Internet applications are deployed to the application servers with a single application server configuration that works for all locales. Figure 8–2 shows the architecture of a multilingual Internet application.

Figure 8–2 Multilingual Internet Application Architecture



To support multiple locales in a single application instance, the application may need to do the following:

- Dynamically detect the locale of the users and adapt to the locale by constructing HTML pages in the language and cultural conventions of the locale

- Process character data in Unicode so that data in any language can be supported. Character data can be entered by users or retrieved from back-end databases.
- Dynamically determine the HTML page encoding (or character set) to be used for HTML pages and convert content from Unicode to the page encoding and the reverse.

The following are major advantages of deploying multilingual Internet application:

- Using a single application server configuration for all application servers simplifies the deployment configuration and hence reduces the cost of maintenance.
- Performance tuning and capacity planning do not depend on the number of locales supported by the Web site.
- Introducing additional locales is relatively easy. No extra machines are necessary for the new locales.
- Testing the application across different locales can be done in a single testing environment.
- This model can support multilingual content within the same instance of the application. For example, a web page containing Japanese, Chinese, English and Arabic data can be easily supported in this model.

The disadvantage of deploying multilingual Internet applications is that it requires extra coding during application development to handle dynamic locale detection and Unicode, which is costly when only one or two languages need to be supported.

Deploying multilingual Internet applications is more appropriate than deploying monolingual applications when Web sites support multiple locales.

Developing a Global Internet Application

Building an Internet application that supports different locales requires good development practices.

For multilingual Internet applications, the application itself must be aware of the user's locale and be able to present locale-appropriate content to the user. Clients must be able to communicate with the application server regardless of the client's locale. The application server then communicates with the database server, exchanging data while maintaining the preferences of the different locales and character set settings. One of the main considerations when developing a multilingual Internet application is to be able to dynamically detect, cache, and provide the appropriate contents according to the user's preferred locale.

For monolingual Internet applications, the locale of the user is always fixed and usually follows the default locale of the run-time environment. Hence, the locale configuration is much simpler.

The following sections describe some of the most common issues that developers encounter when building a global Internet application:

- Locale Determination
- Locale Awareness
- Localizing the Content

Locale Determination

To be locale-aware or locale-sensitive, Internet applications must be able to determine the preferred locale of the user.

Monolingual applications always serve users with the same locale, and that locale should be equivalent to the default run-time locale of the corresponding programming environment.

Multilingual applications can determine a user locale dynamically in three ways. Each method has advantages and disadvantages, but they can be used together in the applications to complement each other. The user locale can be determined in the following ways:

- Based on the user profile information from a LDAP directory server such as the Oracle Internet Directory or other user profile tables stored inside the database

  The schema for the user profile should include preferred locale attribute to indicate the locale of a user. This way of determining a locale user does not work if a user has not been logged on before.

- Based on the default locale of the browser

  Get the default ISO locale setting from a browser. The default ISO locale of the browser is sent through the Accept-Language HTTP header in every HTTP request. If the Accept-Language header is `NULL`, then the desired locale should default to English. The drawback of this approach is that the Accept-Language header may not be a reliable source of information for the locale of a user.

- Based on user selection

  Allow users to select a locale from a list box or from a menu, and switch the application locale to the one selected.

The Globalization Development Kit provides an application framework that enables you to use these locale determination methods declaratively.

See Also: "Getting Started with the Globalization Development Kit" on page 8-7

Locale Awareness

To be locale-aware or locale-sensitive, Internet applications need to determine the locale of a user. After the locale of a user is determined, applications should:

- Construct HTML content in the language of the locale
- Use the cultural conventions implied by the locale

Locale-sensitive functions, such as date, time, and monetary formatting, are built into various programming environments such as Java and PL/SQL. Applications may use them to format the HTML pages according to the cultural conventions of the locale of a user. A locale is represented differently in different programming environments. For example, the French (Canada) locale is represented in different environments as follows:

- In the ISO standard, it is represented by `fr-CA` where `fr` is the language code defined in the ISO 639 standard and `CA` is the country code defined in the ISO 3166 standard.
- In Java, it is represented as a Java locale object constructed with `fr`, the ISO language code for French, as the language and `CA`, the ISO country code for Canada, as the country. The Java locale name is `fr_CA`.

- In PL/SQL and SQL, it is represented mainly by the NLS_LANGUAGE and NLS_TERRITORY session parameters where the value of the NLS_LANGUAGE parameter is equal to CANADIAN FRENCH and the value of the NLS_TERRITORY parameter is equal to CANADA.

If you write applications for more than one programming environment, then locales must be synchronized between environments. For example, Java applications that call PL/SQL procedures should map the Java locales to the corresponding NLS_LANGUAGE and NLS_TERRITORY values and change the parameter values to match the user's locale before calling the PL/SQL procedures.

The Globalization Development Kit for Java provides a set of Java classes to ensure consistency on locale-sensitive behaviors with Oracle databases.

Localizing the Content

For the application to support a multilingual environment, it must be able to present the content in the preferred language and in the locale convention of the user. Hard-coded user interface text must first be externalized from the application, together with any image files, so that they can be translated into the different languages supported by the application. The translation files then must be staged in separate directories, and the application must be able to locate the relevant content according to the user locale setting. Special application handling may also be required to support a fallback mechanism, so that if the user-preferred locale is not available, then the next most suitable content is presented. For example, if Canadian French content is not available, then it may be suitable for the application to switch to the French files instead.

Getting Started with the Globalization Development Kit

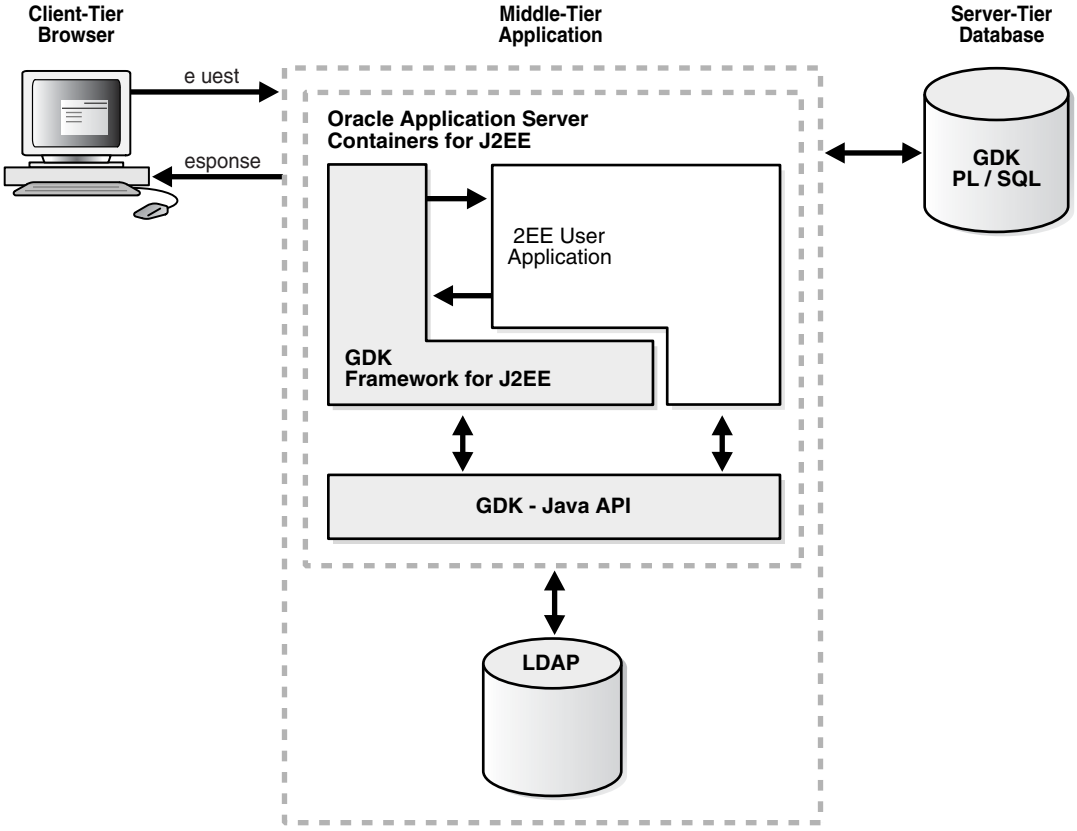
The Globalization Development Kit (GDK) for Java provides a J2EE application framework and Java APIs to develop globalized Internet applications using the best globalization practices and features designed by Oracle. It reduces the complexities and simplifies the code that Oracle developers require to develop globalized Java applications.

GDK for Java complements the existing globalization features in J2EE. Although the J2EE platform already provides a strong foundation for building globalized applications, its globalization functionalities and behaviors can be quite different from Oracle's functionalities. GDK for Java provides synchronization of locale-sensitive behaviors between the middle-tier Java application and the database server.

GDK for PL/SQL contains a suite of PL/SQL packages that provide additional globalization functionalities for applications written in PL/SQL.

Figure 8–3 shows the major components of the GDK and how they are related to each other. User applications run on the J2EE container of Oracle Application Server in the middle tier. GDK provides the application framework that the J2EE application uses to simplify coding to support globalization. Both the framework and the application call the GDK Java API to perform locale-sensitive tasks. GDK for PL/SQL offers PL/SQL packages that help to resolve globalization issues specific to the PL/SQL environment.

Figure 8–3 GDK Components



The functionalities offered by GDK for Java can be divided into two categories:

- The GDK application framework for J2EE provides the globalization framework for building J2EE-based Internet application. The framework encapsulates the complexity of globalization programming, such as determining user locale, maintaining locale persistency, and processing locale information. It consists of a set of Java classes through which applications can gain access to the framework. These associated Java classes enable applications to code against the framework so that globalization behaviors can be extended declaratively.
- The GDK Java API offers development support in Java applications and provides consistent globalization operations as provided in Oracle database servers. The API is accessible and is independent of the GDK framework so that standalone Java applications and J2EE applications that are not based on the GDK framework are able to access the individual features offered by the Java API. The features provided in the Java API include data and number formatting, sorting, and handling character sets in the same way as the Oracle Database.

> **Note:** The GDK Java API is supported with JDK versions 1.4 and later.

GDK for Java is contained in nine `.jar` files, all in the form of `orai18n*jar`. These files are shipped with the Oracle Database, in the `$ORACLE_HOME/jlib` directory. If the application using the GDK is not hosted on the same machine as the database, then the GDK files must be copied to the application server and included into the `CLASSPATH` to run your application. You do not need to install the Oracle Database into your

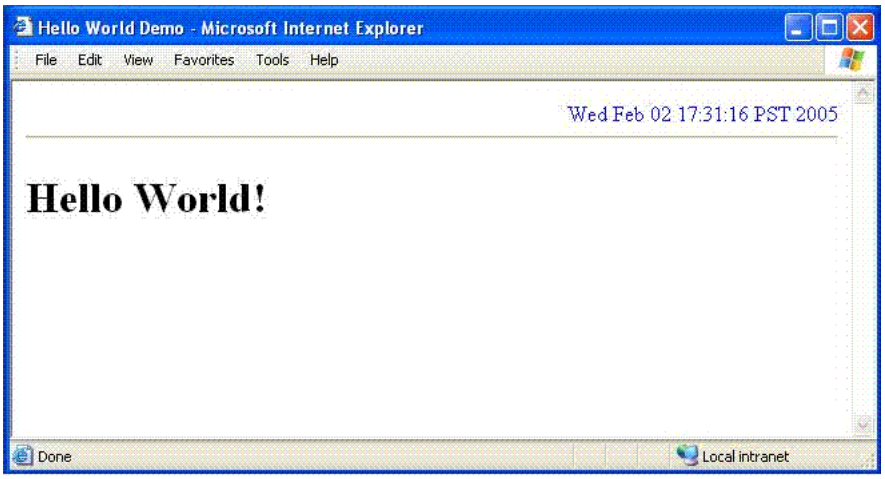
application server to be able to run the GDK inside your Java application. GDK is a pure Java library that runs on every platform. The Oracle client parameters NLS_LANG and ORACLE_HOME are not required.

GDK Quick Start

This section explains how to modify a monolingual application to be a global, multilingual application using GDK. The subsequent sections in this chapter provide detailed information on using GDK.

Figure 8–4 shows a screenshot from a monolingual Web application.

Figure 8–4 Original HelloWorld Web Page



The initial, non-GDK HelloWorld Web application simply prints a "Hello World!" message, along with the current date and time in the top right hand corner of the page. Example 8–1, "HelloWorld JSP Page Code" shows the original HelloWorld JSP source code for the preceding image.

Example 8–1 HelloWorld JSP Page Code

```
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
    <title>Hello World Demo</title>
  </head>
  <body>
  <div style="color: blue;" align="right">
    <%= new java.util.Date(System.currentTimeMillis()) %>
  </div>
  <hr/>
  <h1>Hello World!</h1>
  </body>
</html>
```

Example 8–2, "HelloWorld web.xml Code" shows the corresponding Web application descriptor file for the HelloWorld message.

Example 8–2 HelloWorld web.xml Code

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <description>web.xml file for the monolingual Hello World</description>
  <session-config>
    <session-timeout>35</session-timeout>
  </session-config>
  <mime-mapping>
    <extension>html</extension>
    <mime-type>text/html</mime-type>
  </mime-mapping>
  <mime-mapping>
    <extension>txt</extension>
    <mime-type>text/plain</mime-type>
  </mime-mapping>
</web-app>
```

The HelloWorld JSP code in Example 8–1 is only for English-speaking users. Some of the problems with this code are as follows:

- There is no locale determination based on user preference or browser setting.
- The title and the heading are included in the code.
- The date and time value is not localized based on any locale preference.
- The character encoding included in the code is for Latin-1.

The GDK framework can be integrated into the HelloWorld code to make it a global, multilingual application. The preceding code can be modified to include the following features:

- Automatic locale negotiation to detect the user's browser locale and serve the client with localized HTML pages. The supported application locales are configured in the GDK configuration file.
- Locale selection list to map the supported application locales. The list can have application locale display names which are the name of the country representing the locale. The list will be included on the Web page so users can select a different locale.
- GDK framework and API for globalization support for the HelloWorld JSP. This involves selecting display strings in a locale-sensitive manner and formatting the date and time value.

Modifying the HelloWorld Application

This section explains how to modify the HelloWorld application to support globalization. The application will be modified to support three locales, Simplified Chinese (zh-CN), Swiss German (de-CH), and American English (en-US). The following rules will be used for the languages:

- If the client locale supports one of these languages, then that language will be used for the application.
- If the client locale does not support one of these languages, then American English will be used for the application.

In addition, the user will be able to change the language by selecting a supported locales from the locale selection list. The following tasks describe how to modify the application:

- Task 1: Enable the Hello World Application to use the GDK Framework
- Task 2: Configure the GDK Framework for Hello World
- Task 3: Enable the JSP or Java Servlet
- Task 4: Create the Locale Selection List
- Task 5: Build the Application

Task 1: Enable the Hello World Application to use the GDK Framework

In this task, the GDK filter and a listener are configured in the Web application deployment descriptor file, `web.xml`. This allows the GDK framework to be used with the HelloWorld application. Example 8–3 shows the GDK-enabled `web.xml` file.

Example 8–3 The GDK-enabled web.xml File

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <description>web.xml file for Hello World</description>
   <!-- Enable the application to use the GDK Application Framework.-->
  <filter>
    <filter-name>GDKFilter</filter-name>
    <filter-class>oracle.i18n.servlet.filter.ServletFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>GDKFilter</filter-name>
    <url-pattern>*.jsp</url-pattern>
  </filter-mapping>

  <listener>
    <listener-class>oracle.i18n.servlet.listener.ContextListener</listener-class>
  </listener>

  <session-config>
    <session-timeout>35</session-timeout>
  </session-config>
  <mime-mapping>
    <extension>html</extension>
    <mime-type>text/html</mime-type>
  </mime-mapping>
  <mime-mapping>
    <extension>txt</extension>
    <mime-type>text/plain</mime-type>
  </mime-mapping>
</web-app>
```

The following tags were added to the file:

- `<filter>`

  The filter name is GDKFilter, and the filter class is `oracle.i18n.servlet.filter.ServletFilter`.

- `<filter-mapping>`

The GDKFilter is specified in the tag, as well as the URL pattern.

- `<listener>`

  The listener class is `oracle.i18n.servlet.listener.ContextListener`. The default GDK listener is configured to instantiate GDK ApplicationContext, which controls application scope operations for the framework.

Task 2: Configure the GDK Framework for Hello World

The GDK application framework is configured with the application configuration file `gdkapp.xml`. The configuration file is located in the same directory as the `web.xml` file. Example 8–4 shows the `gdkapp.xml` file.

Example 8–4 GDK Configuration File gdkapp.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<gdkapp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="gdkapp.xsd">

  <!-- The Hello World GDK Configuration -->
  <page-charset default="yes">UTF-8</page-charset>

  <!-- The supported application locales for the Hello World Application -->

  <application-locales>
    <locale>de-CH</locale>
    <locale default="yes">en-US</locale>
    <locale>zh-CN</locale>
  </application-locales>

  <locale-determine-rule>
    <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>

<locale-source>oracle.i18n.servlet.localesource.HttpAcceptLanguage</locale-source>
  </locale-determine-rule>

  <message-bundles>
    <resource-bundle name="default">com.oracle.demo.Messages</resource-bundle>
  </message-bundles>
</gdkapp>
```

The file must be configured for J2EE applications. The following tags are used in the file:

- `<page-charset>`

  The page encoding tag specifies the character set used for HTTP requests and responses. The UTF-8 encoding is used as the default because many languages can be represented by this encoding.

- `<application-locales>`

  Configuring the application locales in the `gdkapp.xml` file makes a central place to define locales. This makes it easier to add and remove locales without changing source code. The locale list can be retrieved using the GDK API call `ApplicationContext.getSupportedLocales`.

- `<locale-determine-rule>`

  The language of the initial page is determined by the language setting of the browser. The user can override this language by choosing from the list. The

`locale-determine-rule` is used by GDK to first try the Accept-Language HTTP header as the source of the locale. If the user selects a locale from the list, then the JSP posts a locale parameter value containing the selected locale. The GDK then sends a response with the contents in the selected language.

- `<message-bundles>`

  The message resource bundles allow an application access to localized static content that may be displayed on a Web page. The GDK framework configuration file allows an application to define a default resource bundle for translated text for various languages. In the HelloWorld example, the localized string messages are stored in the Java ListResourceBundle bundle named `Messages`. The `Messages` bundle consists of base resources for the application which are in the default locale. Two more resource bundles provide the Chinese and German translations. These resource bundles are named `Messages_zh_CN.java` and `Messages_de.java` respectively. The HelloWorld application will select the right translation for "Hello World!" from the resource bundle based on the locale determined by the GDK framework. The `<message-bundles>` tag is used to configure the resource bundles that the application will use.

Task 3: Enable the JSP or Java Servlet

JSPs and Java servlets must be enabled to use the GDK API. Example 8–5 shows a JSP that has been modified to enable to use the GDK API and services. This JSP can accommodate any language and locale.

Example 8–5 Enabled HelloWorld JSP

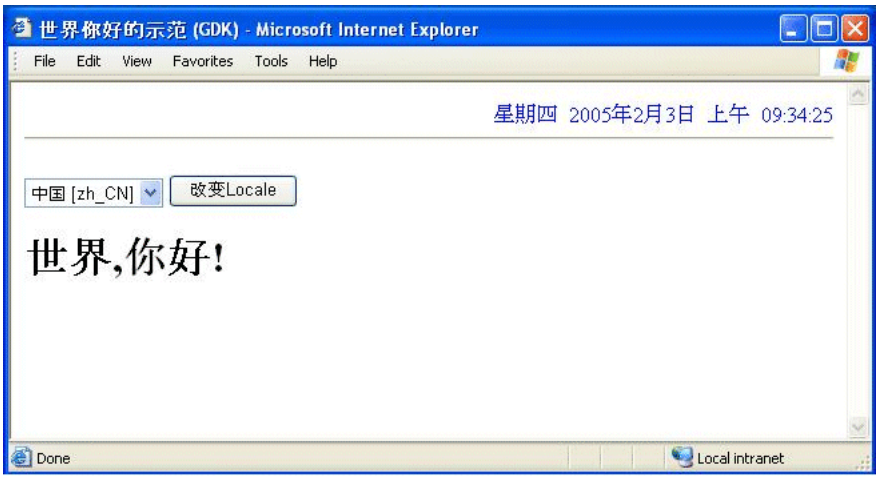
```
. . .
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title><%= localizer.getMessage("helloWorldTitle") %></title>
  </head>

  <body>
  <div style="color: blue;" align="right">
    <% Date currDate= new Date(System.currentTimeMillis()); %>
    <%=localizer.formatDateTime(currDate, OraDateFormat.LONG)%>
  </div>
  <hr/>

  <div align="left">
  <form>
    <select name="locale" size="1">
      <%= getCountryDropDown(request)%>
    </select>
    <input  type="submit" value="<%= localizer.getMessage("changeLocale") %>">
  </input>
  </form>
  </div>
  <h1><%= localizer.getMessage("helloWorld") %></h1>
  </body>
</html>
```

Figure 8–5 shows the HelloWorld application that has been configured with the zh-CN locale as the primary locale for the browser preference. The HelloWorld string and page title are displayed in Simplified Chinese. In addition, the date is formatted in the zh-CN locale convention. This example allows the user to override the locale from the locale selection list.

Figure 8–5 HelloWorld Localized for the zh-CN Locale



When the locale changes or is initialized using the HTTP Request Accept-Language header or the locale selection list, the GUI behaves appropriately for that locale. This means the date and time value in the upper right corner is localized properly. In addition, the strings are localized and displayed on the HelloWorld page.

The GDK Java Localizer class provides capabilities to localize the contents of a Web page based on the automatic detection of the locale by the GDK framework.

The following code retrieves an instance of the localizer based on the current HTTPServletRequest object. In addition, several imports are declared for use of the GDK API within the JSP page. The localizer retrieves localized strings in a locale-sensitive manner with fallback behavior, and formats the date and time.

```
<%@page contentType="text/html;charset=UTF-8"%>
<%@page import="java.util.*, oracle.i18n.servlet.*" %>
<%@page import="oracle.i18n.util.*, oracle.i18n.text.*" %>

<%
    Localizer localizer = ServletHelper.getLocalizerInstance(request);
%>
```

The following code retrieves the current date and time value stored in the currDate variable. The value is formatted by the localizer `formatDateTime` method. The `OraDateFormat.LONG` parameter in the `formatDateTime` method instructs the localizer to format the date using the locale's long formatting style. If the locale of the incoming request is changed to a different locale with the locale selection list, then the date and time value will be formatted according to the conventions of the new locale. No code changes need to be made to support newly-introduced locales.

```
div style="color: blue;" align="right">

    <% Date currDate= new Date(System.currentTimeMillis()); %>
    <%=localizer.formatDateTime(currDate, OraDateFormat.LONG)%>
  </div>
```

The HelloWorld JSP can be reused for any locale because the HelloWorld string and title are selected in a locale-sensitive manner. The translated strings are selected from a resource bundle.

The GDK uses the OraResourceBundle class for implementing the resource bundle fallback behavior. The following code shows how the Localizer picks the HelloWorld message from the resource bundle.

The default application resource bundle Messages is declared in the `gdkapp.xml` file. The localizer uses the message resource bundle to pick the message and apply the locale-specific logic. For example, if the current locale for the incoming request is "de-CH", then the message will first be looked for in the messages_de_CH bundle. If it does not exist, then it will look up in the Messages_de resource bundle.

```
<h1><%= localizer.getMessage("helloWorld") %></h1>
```

Task 4: Create the Locale Selection List

The locale selection list is used to override the selected locale based on the HTTP Request Accept-Language header. The GDK framework checks the locale parameter passed in as part of the HTTP POST request as a value for the new locale. A locale selected with the locale selection list is posted as the locale parameter value. GDK uses this value for the request locale. All this happens implicitly within the GDK code.

The following code sample displays the locale selection list as an HTML select tag with the name locale. The submit tag causes the new value to be posted to the server. The GDK framework retrieves the correct selection.

```
<form>
    <select name="locale" size="1">
      <%= getCountryDropDown(request)%>
    </select>
    <input type="submit" value="<%= localizer.getMessage("changeLocale") %>">
    </input>
</form>
```

The locale selection list is constructed from the HTML code generated by the `getCountryDropDown` method. The method converts the configured application locales into localized country names.

A call is made to the ServletHelper class to get the ApplicationContext object associated with the current request. This object provides the globalization context for an application, which includes information such as supported locales and configuration information. The `getSupportedLocales` call retrieves the list of locales in the `gdkapp.xml` file. The configured application locale list is displayed as options of the HTML select. The `OraDisplayLocaleInfo` class is responsible for providing localization methods of locale-specific elements such as country and language names.

An instance of this class is created by passing in the current locale automatically determined by the GDK framework. GDK creates requests and response wrappers for HTTP request and responses. The `request.getLocale()` method returns the GDK determined locale based on the locale determination rules.

The `OraDsiplayLocaleInfo.getDisplayCountry` method retrieves the localized country names of the application locales. An HTML option list is created in the ddOptBuffer string buffer. The `getCountryDropDown` call returns a string containing the following HTML values:

```
<option value="en_US" selected>United States [en_US]</option>
<option value="zh_CN">China [zh_CN]</option>
<option value="de_CH">Switzerland [de_CH]</option>
```

In the preceding values, the en-US locale is selected for the locale. Country names are generated are based on the current locale.

Example 8–6 shows the code for constructing the locale selection list.

Example 8–6 Constructing the Locale Selection List

```
<%!
    public String getCountryDropDown(HttpServletRequest request)
    {
        StringBuffer ddOptBuffer=new StringBuffer();
        ApplicationContext ctx =
ServletHelper.getApplicationContextInstance(request);
        Locale[] appLocales = ctx.getSupportedLocales();
        Locale currentLocale = request.getLocale();

        if (currentLocale.getCountry().equals(""))
        {
             // Since the Country was not specified get the Default Locale
             // (with Country) from the GDK
             OraLocaleInfo oli = OraLocaleInfo.getInstance(currentLocale);
             currentLocale = oli.getLocale();
        }

        OraDisplayLocaleInfo odli =
OraDisplayLocaleInfo.getInstance(currentLocale);
        for (int i=0;i<appLocales.length; i++)
        {
            ddOptBuffer.append("<option value=\"" + appLocales[i] + "\"" +
            (appLocales[i].getLanguage().equals(currentLocale.getLanguage()) ? "
selected" : "") +
                    ">" + odli.getDisplayCountry(appLocales[i]) +
                " [" +  appLocales[i] + "]</option>\n");
        }

        return ddOptBuffer.toString();
    }
%>
```

Task 5: Build the Application

In order to build the application, the following files must be specified in the classpath:

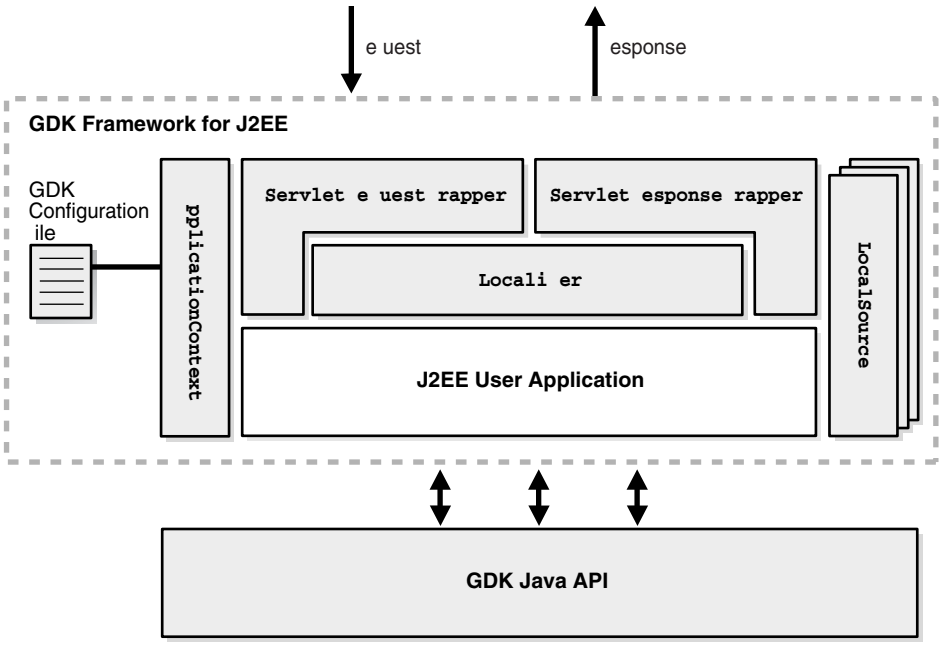
- `orai18n.jar`
- `regexp.jar`

The `orai18n.jar` file contains the GDK framework and the API. The `regexp.jar` file contains the regular expression library. The GDK API also has locale determination capabilities. The classes are supplied by the `ora18n-lcsd.jar` file.

GDK Application Framework for J2EE

GDK for Java provides the globalization framework for middle-tier J2EE applications. The framework encapsulates the complexity of globalization programming, such as determining user locale, maintaining locale persistency, and processing locale information. This framework minimizes the effort required to make Internet applications global-ready. The GDK application framework is shown in Figure 8–6.

Figure 8–6 GDK Application Framework for J2EE



The main Java classes composing the framework are as follows:

- `ApplicationContext` provides the globalization context of an application. The context information includes the list of supported locales and the rule for determining user-preferred locale. The context information is obtained from the GDK application configuration file for the application.
- The set of `LocaleSource` classes can be plugged into the framework. Each `LocaleSource` class implements the `LocaleSource` interface to get the locale from the corresponding source. Oracle bundles several `LocaleSource` classes in GDK. For example, the `DBLocaleSource` class obtains the locale information of the current user from a database schema. You can also write a customized `LocaleSource` class by implementing the same `LocaleSource` interface and plugging it into the framework.
- `ServletRequestWrapper` and `ServletResponseWrapper` are the main classes of the GDK Servlet filter that transforms HTTP requests and HTTP responses. `ServletRequestWrapper` instantiates a `Localizer` object for each HTTP request based on the information gathered from the `ApplicationContext` and `LocaleSource` objects and ensures that forms parameters are handled properly. `ServletResponseWrapper` controls how HTTP response should be constructed.
- `Localizer` is the all-in-one object that exposes the important functions that are sensitive to the current user locale and application context. It provides a centralized set of methods for you to call and make your applications behave appropriately to the current user locale and application context.
- The GDK Java API is always available for applications to enable finer control of globalization behavior.

The GDK application framework simplifies the coding required for your applications to support different locales. When you write a J2EE application according to the application framework, the application code is independent of what locales the application supports, and you control the globalization support in the application by

defining it in the GDK application configuration file. There is no code change required when you add or remove a locale from the list of supported application locales.

The following list gives you some idea of the extent to which you can define the globalization support in the GDK application configuration file:

- You can add and remove a locale from the list of supported locales.
- You can change the way the user locale is determined.
- You can change the HTML page encoding of your application.
- You can specify how the translated resources can be located.
- You can plug a new `LocaleSource` object into the framework and use it to detect a user locale.

This section includes the following topics:

Making the GDK Framework Available to J2EE Applications

The behavior of the GDK application framework for J2EE is controlled by the GDK application configuration file, `gdkapp.xml`. The application configuration file allows developers to specify the behaviors of globalized applications in one centralized place. One application configuration file is required for each J2EE application using the GDK. The `gdkapp.xml` file should be placed in the `./WEB-INF` directory of the J2EE environment of the application. The file dictates the behavior and the properties of the GDK framework and the application that is using it. It contains locale mapping tables, character sets of content files, and globalization parameters for the configuration of the application. The application administrator can modify the application configuration file to change the globalization behavior in the application, without needing to change the programs and to recompile them.

See Also: "The GDK Application Configuration File" on page 8-35

For a J2EE application to use the GDK application framework defined by the corresponding GDK application configuration file, the GDK Servlet filter and the GDK context listener must be defined in the `web.xml` file of the application. The `web.xml` file should be modified to include the following at the beginning of the file:

```
<web-app>
<!-- Add GDK filter that is called after the authentication -->

<filter>
    <filter-name>gdkfilter</filter-name>
    <filter-class>oracle.i18n.servlet.filter.ServletFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>gdkfilter</filter-name>
    <url-pattern>*.jsp</url-pattern>
```

```
</filter-mapping>

<!-- Include the GDK context listener -->

 <listener>
<listener-class>oracle.i18n.servlet.listener.ContextListener</listener-class>
 </listener>
</web-app>
```

Examples of the `gdkapp.xml` and `web.xml` files can be found in the `$ORACLE_HOME/nls/gdk/demo` directory.

The GDK application framework supports Servlet container version 2.3 and later. It uses the Servlet filter facility for transparent globalization operations such as determining the user locale and specifying the character set for content files. The `ContextListener` instantiates GDK application parameters described in the GDK application configuration file. The `ServletFilter` overrides the request and response objects with a GDK request (`ServletRequestWrapper`) and response (`ServletResponseWrapper`) objects, respectively.

If other application filters are used in the application to also override the same methods, then the filter in the GDK framework may return incorrect results. For example, if `getLocale` returns `en_US`, but the result is overridden by other filters, then the result of the GDK locale detection mechanism is affected. All of the methods that are being overridden in the filter of the GDK framework are documented in *Oracle Globalization Development Kit Java API Reference*. Be aware of potential conflicts when using other filters together with the GDK framework.

Integrating Locale Sources into the GDK Framework

Determining the user's preferred locale is the first step in making an application global-ready. The locale detection offered by the J2EE application framework is primitive. It lacks the method that transparently retrieves the most appropriate user locale among locale sources. It provides locale detection by the HTTP language preference only, and it cannot support a multilevel locale fallback mechanism. The GDK application framework provides support for predefined locale sources to complement J2EE. In a web application, several locale sources are available. Table 8–1 summarizes locale sources that are provided by the GDK.

Table 8–1 Locale Resources Provided by the GDK

Locale	Description
HTTP language preference	Locales included in the HTTP protocol as a value of `Accept-Language`. This is set at the web browser level. A locale fallback operation is required if the browser locale is not supported by the application.
User input locale	Locale specified by the user from a menu or a parameter in the HTTP protocol
User profile locale preference from database	Locale preference stored in the database as part of the user profiles
Application default locale	A locale defined in the GDK application configuration file. This locale is defined as the default locale for the application. Typically, this is used as a fallback locale when the other locale sources are not available.

See Also: "The GDK Application Configuration File" on page 8-35 for information about the GDK multilevel locale fallback mechanism

The GDK application framework provides seamless support for predefined locale sources, such as user input locale, HTTP language preference, user profile locale preference in the database, and the application default locale. You can incorporate the locale sources to the framework by defining them under the `<locale-determine-rule>` tag in the GDK application configuration file as follows:

```
<locale-determine-rule>
    <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>
    <locale-source>oracle.i18n.servlet.localesource.HTTPAcceptLanguage</locale-source>
</locale-determine-rule>
```

The GDK framework uses the locale source declaration order and determines whether a particular locale source is available. If it is available, then it is used as the source, otherwise, it tries to find the next available locale source for the list. In the preceding example, if the `UserInput` locale source is available, it is used first, otherwise, the `HTTPAcceptLanguage` locale source will be used.

Custom locale sources, such as locale preference from an LDAP server, can be easily implemented and integrated into the GDK framework. You must implement the `LocaleSource` interface and specify the corresponding implementation class under the `<locale-determine-rule>` tag in the same way as the predefined locale sources were specified.

The `LocaleSource` implementation not only retrieves the locale information from the corresponding source to the framework but also updates the locale information to the corresponding source when the framework tells it to do so. Locale sources can be read-only or read/write, and they can be cacheable or noncacheable. The GDK framework initiates updates only to read/write locale sources and caches the locale information from cacheable locale sources. Examples of custom locale sources can be found in the `$ORACLE_HOME/nls/gdk/demo` directory.

> **See Also:** *Oracle Globalization Development Kit Java API Reference* for more information about implementing a `LocaleSource`

Getting the User Locale From the GDK Framework

The GDK offers automatic locale detection to determine the current locale of the user. For example, the following code retrieves the current user locale in Java. It uses a `Locale` object explicitly.

```
Locale loc = request.getLocale();
```

The `getLocale()` method returns the `Locale` that represents the current locale. This is similar to invoking the `HttpServletRequest.getLocale()` method in JSP or Java Servlet code. However, the logic in determining the user locale is different, because multiple locale sources are being considered in the GDK framework.

Alternatively, you can get a `Localizer` object that encapsulates the `Locale` object determined by the GDK framework. For the benefits of using the `Localizer` object, see "Implementing Locale Awareness Using the GDK Localizer" on page 8-21.

```
Localizer localizer = ServletHelper.getLocalizerInstance(request);
Locale loc = localizer.getLocale();
```

The locale detection logic of the GDK framework depends on the locale sources defined in the GDK application configuration file. The names of the locale sources are registered in the application configuration file. The following example shows the locale determination rule section of the application configuration file. It indicates that the user-preferred locale can be determined from either the LDAP server or from the HTTP `Accept-Language` header. The `LDAPUserSchema` locale source class should be

provided by the application. Note that all of the locale source classes have to be extended from the LocaleSource abstract class.

```
<locale-determine-rule>
    <locale-source>LDAPUserSchema</locale-source>
    <locale-source>oracle.i18n.localesource.HTTPAcceptLanguage</locale-source>
</locale-determine-rule>
```

For example, when the user is authenticated in the application and the user locale preference is stored in an LDAP server, then the LDAPUserSchema class connects to the LDAP server to retrieve the user locale preference. When the user is anonymous, then the HttpAcceptLanguage class returns the language preference of the web browser.

The cache is maintained for the duration of a HTTP session. If the locale source is obtained from the HTTP language preference, then the locale information is passed to the application in the HTTP Accept-Language header and not cached. This enables flexibility so that the locale preference can change between requests. The cache is available in the HTTP session.

The GDK framework exposes a method for the application to overwrite the locale preference information persistently stored in locale sources such as the LDAP server or the user profile table in the database. This method also resets the current locale information stored inside the cache for the current HTTP session. The following is an example of overwriting the preferred locale using the store command.

```
<input type="hidden"
name="<%=appctx.getParameterName(LocaleSource.Parameter.COMMAND)%>"
value="store">
```

To discard the current locale information stored inside the cache, the clean command can be specified as the input parameter. The following table shows the list of commands supported by the GDK:

Command	Functionality
store	Updates user locale preferences in the available locale sources with the specified locale information. This command is ignored by the read-only locale sources.
clean	Discards the current locale information in the cache.

Note that the GDK parameter names can be customized in the application configuration file to avoid name conflicts with other parameters used in the application.

Implementing Locale Awareness Using the GDK Localizer

The Localizer object obtained from the GDK application framework is an all-in-one globalization object that provides access to functions that are commonly used in building locale awareness in your applications. In addition, it provides functions to get information about the application context, such as the list of supported locales. The Localizer object simplifies and centralizes the code required to build consistent locale awareness behavior in your applications.

The oracle.i18n.servlet package contains the Localizer class. You can get the Localizer instance as follows:

```
Localizer lc = ServletHelper.getLocalizerInstance(request);
```

The Localizer object encapsulates the most commonly used locale-sensitive information determined by the GDK framework and exposes it as locale-sensitive methods. This object includes the following functionalities pertaining to the user locale:

- Format date in long and short formats
- Format numbers and currencies
- Get collation key value of a string
- Get locale data such as language, country and currency names
- Get locale data to be used for constructing user interface
- Get a translated message from resource bundles
- Get text formatting information such as writing direction
- Encode and decode URLs
- Get the common list of time zones and linguistic sorts

For example, when you want to display a date in your application, you may want to call the Localizer.formatDate() or Localizer.formateDateTime() methods. When you want to determine the writing direction of the current locale, you can call the Localizer.getWritingDirection() and Localizer.getAlignment() to determine the value used in the <DIR> tag and <ALIGN> tag respectively.

The Localizer object also exposes methods to enumerate the list of supported locales and their corresponding languages and countries in your applications.

The Localizer object actually makes use of the classes in the GDK Java API to accomplish its tasks. These classes include, but are not limited to, the following: OraDateFormat, OraNumberFormat, OraCollator, OraLocaleInfo, oracle.i18n.util.LocaleMapper, oracle.i18n.net.URLEncoder, and oracle.i18n.net.URLDecoder.

The Localizer object simplifies the code you need to write for locale awareness. It maintains caches of the corresponding objects created from the GDK Java API so that the calling application does not need to maintain these objects for subsequent calls to the same objects. If you require more than the functionality the Localizer object can provide, then you can always call the corresponding methods in the GDK Java API directly.

Note: Strings returned by many Localizer methods, such as formatted dates and locale-specific currency symbols, depend on locale data that may be provided by users through URLs or form input. For example, the locale source class oracle.i18n.servlet.localesource.UserInput provides various datetime format patterns and the ISO currency abbreviation retrieved from a page URL. A datetime format pattern may include double-quoted literal strings with arbitrary contents. To prevent cross-site script injection attacks, strings returned by Localizer methods must be properly escaped before being displayed as part of an HTML page, for example, by applying the method encode of the class oracle.i18n.net.CharEntityReference.

See Also: *Oracle Globalization Development Kit Java API Reference* for detailed information about the Localizer object

Defining the Supported Application Locales in the GDK

The number of locales and the names of the locales that an application needs to support are based on the business requirements of the application. The names of the locales that are supported by the application are registered in the application configuration file. The following example shows the application locales section of the application configuration file. It indicates that the application supports German (`de`), Japanese (`ja`), and English for the US (`en-US`), with English defined as the default fallback application locale. Note that the locale names are based on the IANA convention.

```
<application-locales>
    <locale>de</locale>
    <locale>ja</locale>
    <locale default="yes">en-US</locale>
</application-locales>
```

When the GDK framework detects the user locale, it verifies whether the locale that is returned is one of the supported locales in the application configuration file. The verification algorithm is as follows:

1. Retrieve the list of supported application locales from the application configuration file.
2. Check whether the locale that was detected is included in the list. If it is included in the list, then use this locale as the current client's locale.
3. If there is a variant in the locale that was detected, then remove the variant and check whether the resulting locale is in the list. For example, the Java locale `de_DE_EURO` has a `EURO` variant. Remove the variant so that the resulting locale is `de_DE`.
4. If the locale includes a country code, then remove the country code and check whether the resulting locale is in the list. For example, the Java locale `de_DE` has a country code of `DE`. Remove the country code so that the resulting locale is `de`.
5. If the detected locale does not match any of the locales in the list, then use the default locale that is defined in the application configuration file as the client locale.

By performing steps 3 and 4, the application can support users with the same language requirements but with different locale settings than those defined in the application configuration file. For example, the GDK can support `de-AT` (the Austrian variant of German), `de-CH` (the Swiss variant of German), and `de-LU` (the Luxembourgian variant of German) locales.

The locale fallback detection in the GDK framework is similar to that of the Java Resource Bundle, except that it is not affected by the default locale of the Java VM. This exception occurs because the Application Default Locale can be used during the GDK locale fallback operations.

If the application-locales section is omitted from the application configuration file, then the GDK assumes that the common locales, which can be returned from the `OraLocaleInfo.getCommonLocales` method, are supported by the application.

Handling Non-ASCII Input and Output in the GDK Framework

The character set (or character encoding) of an HTML page is a very important piece of information to a browser and an Internet application. The browser needs to interpret this information so that it can use correct fonts and character set mapping tables for displaying pages. The Internet applications need to know so they can safely process input data from a HTML form based on the specified encoding.

The page encoding can be translated as the character set used for the locale to which an Internet application is serving.

In order to correctly specify the page encoding for HTML pages without using the GDK framework, Internet applications must:

1. Determine the desired page input data character set encoding for a given locale.
2. Specify the corresponding encoding name for each HTTP request and HTTP response.

Applications using the GDK framework can ignore these steps. No application code change is required. The character set information is specified in the GDK application configuration file. At run time, the GDK automatically sets the character sets for the request and response objects. The GDK framework does not support the scenario where the incoming character set is different from that of the outgoing character set.

The GDK application framework supports the following scenarios for setting the character sets of the HTML pages:

- A single local character set is dedicated to the whole application. This is appropriate for a monolingual Internet application. Depending on the properties of the character set, it may be able to support more than one language. For example, most Western European languages can be served by ISO-8859-1.
- Unicode UTF-8 is used for all contents regardless of the language. This is appropriate for a multilingual application that uses Unicode for deployment.
- The native character set for each language is used. For example, English contents are represented in ISO-8859-1, and Japanese contents are represented in Shift_JIS. This is appropriate for a multilingual Internet application that uses a default character set mapping for each locale. This is useful for applications that need to support different character sets based on the user locales. For example, for mobile applications that lack Unicode fonts or Internet browsers that cannot fully support Unicode, the character sets must to be determined for each request.

The character set information is specified in the GDK application configuration file. The following is an example of setting UTF-8 as the character set for all the application pages.

```
<page-charset>UTF-8</page-charset>
```

The page character set information is used by the `ServletRequestWrapper` class, which sets the proper character set for the request object. It is also used by the `ContentType` HTTP header specified in the `ServletResponseWrapper` class for output when instantiated. If `page-charset` is set to `AUTO-CHARSET`, then the character set is assumed to be the default character set for the current user locale. Set `page-charset` to `AUTO-CHARSET` as follows:

```
<page-charset>AUTO-CHARSET</page-charset>
```

The default mappings are derived from the `LocaleMapper` class, which provides the default IANA character set for the locale name in the GDK Java API.

Table 8–2 lists the mappings between the common ISO locales and their IANA character sets.

Table 8–2 Mapping Between Common ISO Locales and IANA Character Sets

ISO Locale	NLS_LANGUAGE Value	NLS_TERRITORY Value	IANA Character Set
ar-SA	ARABIC	SAUDI ARABIA	WINDOWS-1256
de-DE	GERMAN	GERMANY	WINDOWS-1252
en-US	AMERICAN	AMERICA	WINDOWS-1252
en-GB	ENGLISH	UNITED KINGDOM	WINDOWS-1252
el	GREEK	GREECE	WINDOWS-1253
es-ES	SPANISH	SPAIN	WINDOWS-1252
fr	FRENCH	FRANCE	WINDOWS-1252
fr-CA	CANADIAN FRENCH	CANADA	WINDOWS-1252
iw	HEBREW	ISRAEL	WINDOWS-1255
ko	KOREAN	KOREA	EUC-KR
ja	JAPANESE	JAPAN	SHIFT_JIS
it	ITALIAN	ITALY	WINDOWS-1252
pt	PORTUGUESE	PORTUGAL	WINDOWS-1252
pt-BR	BRAZILIAN PORTUGUESE	BRAZIL	WINDOWS-1252
tr	TURKISH	TURKEY	WINDOWS-1254
nl	DUTCH	THE NETHERLANDS	WINDOWS-1252
zh	SIMPLIFIED CHINESE	CHINA	GBK
zh-TW	TRADITIONAL CHINESE	TAIWAN	BIG5

The locale to character set mapping in the GDK can also be customized. To override the default mapping defined in the GDK Java API, a locale-to-character-set mapping table can be specified in the application configuration file.

```
<locale-charset-maps>
   <locale-charset>
      <locale>ja</locale><charset>EUC-JP</charset>
   </locale-charset>
</locale-charset-maps>
```

The previous example shows that for locale Japanese (`ja`), the GDK changes the default character set from SHIFT_JIS to EUC-JP.

See Also: "Oracle Locale Information in the GDK" on page 8-28

Managing Localized Content in the GDK

This section includes the following topics:

- Managing Localized Content in JSPs and Java Servlets
- Managing Localized Content in Static Files

Managing Localized Content in JSPs and Java Servlets

Resource bundles enable access to localized contents at run time in J2SE. Translatable strings within Java servlets and Java Server Pages (JSPs) are externalized into Java resource bundles so that these resource bundles can be translated independently into

different languages. The translated resource bundles carry the same base class names as the English bundles, using the Java locale name as the suffix.

To retrieve translated data from the resource bundle, the `getBundle()` method must be invoked for every request.

```
<% Locale user_locale=request.getLocale();
   ResourceBundle rb=ResourceBundle.getBundle("resource",user_locale); %>
<%= rb.getString("Welcome") %>
```

The GDK framework simplifies the retrieval of text strings from the resource bundles. `Localizer.getMessage()` is a wrapper to the resource bundle.

```
<% Localizer.getMessage ("Welcome") %>
```

Instead of specifying the base class name as `getBundle()` in the application, you can specify the resource bundle in the application configuration file, so that the GDK automatically instantiates a `ResourceBundle` object when a translated text string is requested.

```
<message-bundles>
  <resource-bundle name="default">resource</resource-bundle>
</message-bundles>
```

This configuration file snippet declares a default resource bundle whose translated contents reside in the "resource" Java bundle class. Multiple resource bundles can be specified in the configuration file. To access a nondefault bundle, specify the `name` parameter in the `getMessage` method. The message bundle mechanism uses the `OraResourceBundle` GDK class for its implementation. This class provides the special locale fallback behaviors on top of the Java behaviors. The rules are as follows:

- If the given locale exactly matches the locale in the available resource bundles, it will be used.
- If the resource bundle for Chinese in Singapore (`zh_SG`) is not found, it will fall back to the resource bundle for Chinese in China (`zh_CN`) for Simplified Chinese translations.
- If the resource bundle for Chinese in Hong Kong (`zh_HK`) is not found, it will fall back to the resource bundle for Chinese in Taiwan (`zh_TW`) for Traditional Chinese translations.
- If the resource bundle for Chinese in Macau (`zh_MO`) is not found, it will fall back to the resource bundle for Chinese in Taiwan (`zh_TW`) for Traditional Chinese translations.
- If the resource bundle for any other Chinese (`zh_` and `zh`) is not found, it will fall back to the resource bundle for Chinese in China (`zh_CN`) for Simplified Chinese translations.
- The default locale, which can be obtained by the `Locale.getDefault()` method, will not be considered in the fallback operations.

For example, assume the default locale is `ja_JP` and the resource handle for it is available. When the resource bundle for `es_MX` is requested, and neither resource bundle for `es` or `es_MX` is provided, the base resource bundle object that does not have a local suffix is returned.

The usage of the `OraResourceBundle` class is similar to the `java.util.ResourceBundle` class, but the `OraResearchBundle` class does not instantiate itself. Instead, the return value of the `getBundle` method is an instance of the subclass of the `java.util.ResourceBundle` class.

Managing Localized Content in Static Files

For a application, which supports only one locale, the URL that has a suffix of `/index.html` typically takes the user to the starting page of the application.

In a globalized application, contents in different languages are usually stored separately, and it is common for them to be staged in different directories or with different file names based on the language or the country name. This information is then used to construct the URLs for localized content retrieval in the application.

The following examples illustrate how to retrieve the French and Japanese versions of the index page. Their suffixes are as follows:

```
/fr/index.html
/ja/index.html
```

By using the `rewriteURL()` method of the `ServletHelper` class, the GDK framework handles the logic to locate the translated files from the corresponding language directories. The `ServletHelper.rewriteURL()` method rewrites a URL based on the rules specified in the application configuration file. This method is used to determine the correct location where the localized content is staged.

The following is an example of the JSP code:

```
<img src="<%="ServletHelper.rewriteURL("image/welcome.jpg", request)%>">
<a href="<%="ServletHelper.rewriteURL("html/welcome.html", request)%>">
```

The URL rewrite definitions are defined in the GDK application configuration file:

```
  <url-rewrite-rule fallback="yes">
    <pattern>(.*)/(a-zA-Z0-9_\]+.)$</pattern>
    <result>$1/$A/$2</result>
  </url-rewrite-rule>
```

The pattern section defined in the rewrite rule follows the regular expression conventions. The result section supports the following special variables for replacing:

- `$L` is used to represent the ISO 639 language code part of the current user locale
- `$C` represents the ISO 3166 country code
- `$A` represents the entire locale string, where the ISO 639 language code and ISO 3166 country code are connected with an underscore character (_)
- $1 to $9 represent the matched substrings

For example, if the current user locale is `ja`, then the URL for the `welcome.jpg` image file is rewritten as `image/ja/welcome.jpg`, and `welcome.html` is changed to `html/ja/welcome.html`.

Both `ServletHelper.rewriteURL()`and `Localizer.getMessage()` methods perform consistent locale fallback operations in the case where the translation files for the user locale are not available. For example, if the online help files are not available for the `es_MX` locale (Spanish for Mexico), but the `es` (Spanish for Spain) files are available, then the methods will select the Spanish translated files as the substitute.

GDK Java API

The globalization features and behaviors in Java are not the same as those offered in Oracle Database. For example, J2SE supports a set of locales and character sets that are different from locales and character sets in Oracle Database. This inconsistency can be confusing for users when their application contains data that is formatted based on

two different conventions. For example, dates that are retrieved from the database are formatted using Oracle conventions, such as number and date formatting and linguistic sort ordering. However, the static application data is typically formatted using Java locale conventions. The globalization functionalities in Java can also be different depending on the version of the JDK on which the application runs.

Before Oracle Database 10*g*, when an application was required to incorporate Oracle globalization features, it had to make connections to the database and issue SQL statements. Such operations make the application complicated and generate more network connections to the database.

In Oracle Database 10*g* and later, the GDK Java API extends the globalization features to the middle tier. By enabling applications to perform globalization logic such as Oracle date and number formatting and linguistic sorting in the middle tier, the GDK Java API enables developers to eliminate expensive programming logic in the database. The GDK Java API also provides standard compliance for XQuery. This improves the overall application performance by reducing the database processing load, and by decreasing unnecessary network traffic between the application tier and the back end.

The GDK Java API also offers advanced globalization features, such as language and character set detection, and the enumeration of common locale data for a territory or a language (for example, all time zones supported in Canada). These features are not available in most programming platforms. Without the GDK Java API, developers must write business logic to handle these processes inside the application.

The key functionalities of the GDK Java API are as follows:

- Oracle Locale Information in the GDK
- Oracle Locale Mapping in the GDK
- Oracle Character Set Conversion in the GDK
- Oracle Date, Number, and Monetary Formats in the GDK
- Oracle Binary and Linguistic Sorts in the GDK
- Oracle Language and Character Set Detection in the GDK
- Oracle Translated Locale and Time Zone Names in the GDK
- Using the GDK with E-Mail Programs

Oracle Locale Information in the GDK

Oracle locale definitions, which include languages, territories, linguistic sorts, and character sets, are exposed in the GDK Java API. The naming convention that Oracle uses may be different from other vendors. Although many of these names and definitions follow industry standards, some are Oracle-specific, tailored to meet special customer requirements.

`OraLocaleInfo` is an Oracle locale class that includes language, territory, and collator objects. It provides a method for applications to retrieve a collection of locale-related objects for a given locale. Examples include: a full list of the Oracle linguistic sorts available in the GDK, the local time zones defined for a given territory, or the common languages used in a particular territory.

Following are examples of using the `OraLocaleInfo` class:

```
// All Territories supported by GDK
String[] avterr = OraLocaleInfo.getAvailableTerritories();
```

```
// Local TimeZones for a given Territory

OraLocaleInfo oloc = OraLocaleInfo.getInstance("English", "Canada");
TimeZone[] loctz = oloc.getLocalTimeZones();
```

Oracle Locale Mapping in the GDK

The GDK Java API provides the `LocaleMapper` class. It maps equivalent locales and character sets between Java, IANA, ISO, and Oracle. A Java application may receive locale information from the client that is specified in an Oracle Database locale name or an IANA character set name. The Java application must be able to map to an equivalent Java locale or Java encoding before it can process the information correctly.

The follow example shows using the `LocaleMapper` class.

```
// Mapping from Java locale to Oracle language and Oracle territory

Locale locale = new Locale("it", "IT");
String oraLang = LocaleMapper.getOraLanguage(locale);
String oraTerr = LocaleMapper.getOraTerritory(locale);

// From Oracle language and Oracle territory to Java Locale

locale = LocaleMapper.getJavaLocale("AMERICAN","AMERICA");
locale = LocaleMapper.getJavaLocale("TRADITONAL CHINESE", "");

// From IANA & Java to Oracle Character set

String ocs1     = LocaleMapper.getOraCharacterSet(
                              LocaleMapper.IANA, "ISO-8859-1");
String ocs2     = LocaleMapper.getOraCharacterSet(
                              LocaleMapper.JAVA, "ISO8859_1");
```

The `LocaleMapper` class can also return the most commonly used e-mail character set for a specific locale on both Windows and UNIX platforms. This is useful when developing Java applications that need to process e-mail messages.

See Also: "Using the GDK with E-Mail Programs" on page 8-34

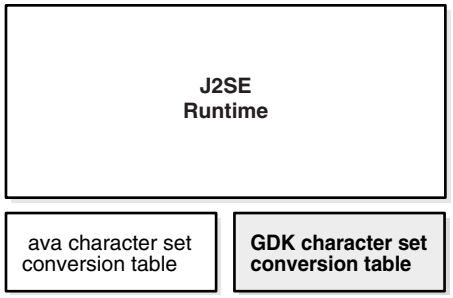
Oracle Character Set Conversion in the GDK

The GDK Java API contains a set of character set conversion classes APIs that enable users to perform Oracle character set conversions. Although Java JDK is already equipped with classes that can perform conversions for many of the standard character sets, they do not support Oracle-specific character sets and Oracle's user-defined character sets.

In JDK 1.4, J2SE introduced an interface for developers to extend Java's character sets. The GDK Java API provides implicit support for Oracle's character sets by using this plug-in feature. You can access the J2SE API to obtain Oracle-specific behaviors.

Figure 8–7 shows that the GDK character set conversion tables are plugged into J2SE in the same way as the Java character set tables. With this pluggable framework of J2SE, the Oracle character set conversions can be used in the same way as other Java character set conversions.

Figure 8–7 Oracle Character Set Plug-In



Because the `java.nio.charset` Java package is not available in JDK versions before 1.4, you must install JDK 1.4 or later to use Oracle's character set plug-in feature.

The GDK character conversion classes support all Oracle character sets including user-defined characters sets. It can be used by Java applications to properly convert to and from Java's internal character set, UTF-16.

Oracle's character set names are proprietary. To avoid potential conflicts with Java's own character sets, all Oracle character set names have an `X-ORACLE-` prefix for all implicit usage through Java's API.

The following is an example of Oracle character set conversion:

```
// Converts the Chinese character "three" from UCS2 to JA16SJIS

String str = "\u4e09";
byte[] barr = str.getBytes("x-oracle-JA16SJIS");
```

Just as with other Java character sets, the character set facility in `java.nio.charset.Charset` is applicable to all of the Oracle character sets. For example, if you want to check whether the specified character set is a superset of another character set, then you can use the `Charset.contains` method as follows:

```
Charset cs1 = Charset.forName("x-oracle-US7ASCII");
Charset cs2 = Charset.forName("x-oracle-WE8WINDOWS1252");
// true if WE8WINDOWS1252 is the superset of US7ASCII, otherwise false.
boolean osc = cs2.contains(cs1);
```

For a Java application that is using the JDBC driver to communicate with the database, the JDBC driver provides the necessary character set conversion between the application and the database. Calling the GDK character set conversion methods explicitly within the application is not required. A Java application that interprets and generates text files based on Oracle's character set encoding format is an example of using Oracle character set conversion classes.

Oracle Date, Number, and Monetary Formats in the GDK

The GDK Java API provides formatting classes that support date, number, and monetary formats using Oracle conventions for Java applications in the `oracle.i18n.text` package.

New locale formats introduced in Oracle Database 10*g*, such as the short and long date, number, and monetary formats, are also exposed in these format classes.

The following are examples of Oracle date, Oracle number, and Oracle monetary formatting:

```
// Obtain the current date and time in the default Oracle LONG format for
```

```
// the locale de_DE (German_Germany)

Locale locale = new Locale("de", "DE");
OraDateFormat odf =
  OraDateFormat.getDateTimeInstance(OraDateFormat.LONG, locale);

// Obtain the numeric value 1234567.89 using the default number format
// for the Locale en_IN (English_India)

locale = new Locale("en", "IN");
OraNumberFormat onf = OraNumberFormat.getNumberInstance(locale);
String nm = onf.format(new Double(1234567.89));

// Obtain the monetary value 1234567.89 using the default currency
// format for the Locale en_US (American_America)

locale = new Locale("en", "US");

onf = OraNumberFormat.getCurrencyInstance(locale);
nm = onf.format(new Double(1234567.89));
```

Oracle Binary and Linguistic Sorts in the GDK

Oracle provides support for binary, monolingual, and multilingual linguistic sorts in the database. In Oracle Database, these sorts provide case-insensitive and accent-insensitive sorting and searching capabilities inside the database. By using the `OraCollator` class, the GDK Java API enables Java applications to sort and search for information based on the latest Oracle binary and linguistic sorting features, including case-insensitive and accent-insensitive options.

Normalization can be an important part of sorting. The composition and decomposition of characters are based on the Unicode standard; therefore, sorting also depends on the Unicode standard. The GDK contains methods to perform composition.

> **Note:** Because each version of the JDK may support a different version of the Unicode standard, the GDK provides an `OraNormalizer` class that is based on the latest version of the standard, which for this release is Unicode 6.2.

The sorting order of a binary sort is based on the Oracle character set that is being used. Except for the UTFE character set, the binary sorts of all Oracle character sets are supported in the GDK Java API. The only linguistic sort that is not supported in the GDK Java API is `JAPANESE`, but a similar and more accurate sorting result can be achieved by using `JAPANESE_M`.

The following example shows string comparisons and string sorting.

Example 8–7 String Comparisons and String Sorting

```
// compares strings using XGERMAN

private static String s1 = "abcSS";
private static String s2 = "abc\u00DF";

String cname = "XGERMAN";
OraCollator ocol = OraCollator.getInstance(cname);
int c = ocol.compare(s1, s2);
```

```
// sorts strings using GENERIC_M

private static String[] source =
  new String[]
  {
    "Hochgeschwindigkeitsdrucker",
    "Bildschirmfu\u00DF",
    "Skjermhengsel",
    "DIMM de Mem\u00F3ria",
    "M\u00F3dulo SDRAM com ECC",
  };


  cname = "GENERIC_M";
  ocol = OraCollator.getInstance(cname);
  List result = getCollationKeys(source, ocol);


private static List getCollationKeys(String[] source, OraCollator ocol)
{
  List karr = new ArrayList(source.length);
  for (int i = 0; i < source.length; ++i)
  {
    karr.add(ocol.getCollationKey(source[i]));
  }
  Collections.sort(karr); // sorting operation
  return karr;
}
```

Oracle Language and Character Set Detection in the GDK

The Oracle Language and Character Set Detection Java classes in the GDK Java API provide a high performance, statistically based engine for determining the character set and language for unspecified text. It can automatically identify language and character set pairs from throughout the world. With each text, the language and character set detection engine sets up a series of probabilities, each probability corresponding to a language and character set pair. The most probable pair statistically identifies the dominant language and character set.

The purity of the text submitted affects the accuracy of the language and character set detection. Only plain text strings are accepted, so any tagging must be stripped before hand. The ideal case is literary text with almost no foreign words or grammatical errors. Text strings that contain a mix of languages or character sets, or nonnatural language text like addresses, phone numbers, and programming language code may yield poor results.

The `LCSDetector` class can detect the language and character set of a byte array, a character array, a string, and an `InputStream` class. It supports both plain text and HTML file detection. It can take the entire input for sampling or only portions of the input for sampling, when the length or both the offset and the length are supplied. For each input, up to three potential language and character set pairs can be returned by the `LCSDetector` class. They are always ranked in sequence, with the pair with the highest probability returned first.

> **See Also:** "Language and Character Set Detection Support" on page A-19 for a list of supported language and character set pairs

The following are examples of using the `LCSDetector` class to enable language and character set detection:

```
// This example detects the character set of a plain text file "foo.txt" and
// then appends the detected ISO character set name to the name of the text file

LCSDetector         lcsd = new LCSDetector();
File             oldfile = new File("foo.txt");
FileInputStream      in = new FileInputStream(oldfile);
lcsd.detect(in);
String          charset = lcsd.getResult().getIANACharacterSet();
File            newfile = new File("foo."+charset+".txt");
oldfile.renameTo(newfile);

// This example shows how to use the LCSDector class to detect the language and
// character set of a byte array

int          offset = 0;
LCSDetector     led = new LCSDetector();
/* loop through the entire byte array */
while ( true )
{
    bytes_read = led.detect(byte_input, offset, 1024);
    if ( bytes_read == -1 )
        break;
    offset += bytes_read;
}
LCSDResultSet    res = led.getResult();

/* print the detection results with close ratios */
System.out.println("the best guess "  );
System.out.println("Langauge " + res.getOraLanguage() );
System.out.println("CharacterSet " + res.getOraCharacterSet() );
int     high_hit = res.getHiHitPairs();
if ( high_hit >= 2 )
{
        System.out.println("the second best guess  " );
        System.out.println("Langauge " + res.getOraLanguage(2) );
        System.out.println("CharacterSet " +res.getOraCharacterSet(2) );
}
if ( high_hit >= 3 )
{
         System.out.println("the third best guess  ");
         System.out.println("Langauge " + res.getOraLanguage(3) );
         System.out.println("CharacterSet " +res.getOraCharacterSet(3) );
}
```

Oracle Translated Locale and Time Zone Names in the GDK

All of the Oracle language names, territory names, character set names, linguistic sort names, and time zone names have been translated into 27 languages including English. They are readily available for inclusion into the user applications, and they provide consistency for the display names across user applications in different languages. `OraDisplayLocaleInfo` is a utility class that provides the translations of locale and attributes. The translated names are useful for presentation in user interface text and for drop-down selection boxes. For example, a native French speaker prefers to select from a list of time zones displayed in French than in English.

The following example shows using `OraDisplayLocaleInfo` to return a list of time zones supported in Canada, using the French translation names.

Example 8–8 Using OraDisplayLocaleInfo to Return a Specific List of Time Zones

```
OraLocaleInfo oloc = OraLocaleInfo.getInstance("CANADIAN FRENCH", "CANADA");
OraDisplayLocaleInfo odloc = OraDisplayLocaleInfo.getInstance(oloc);
TimeZone[] loctzs = oloc.getLocaleTimeZones();
String []  disptz = new string [loctzs.length];
for (int i=0; i<loctzs.length; ++i)
{
  disptz [i]= odloc.getDisplayTimeZone(loctzs[i]);
  ...
}
```

Using the GDK with E-Mail Programs

You can use the GDK `LocaleMapper` class to retrieve the most commonly used e-mail character set. Call `LocaleMapper.getIANACharSetFromLocale`, passing in the locale object. The return value is an array of character set names. The first character set returned is the most commonly used e-mail character set.

The following example illustrates sending an e-mail message containing Simplified Chinese data in the GBK character set encoding.

Example 8–9 Sending E-mail Containing Simplified Chinese Data in GBK Character Set Encoding

```
import oracle.i18n.util.LocaleMapper;
import java.util.Date;
import java.util.Locale;
import java.util.Properties;
import javax.mail.Message;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeUtility;
/**
 * Email send operation sample
 *
 * javac -classpath orai18n.jar:j2ee.jar EmailSampleText.java
 * java  -classpath .:orai18n.jar:j2ee.jar EmailSampleText
 */
public class EmailSampleText
{
  public static void main(String[] args)
  {
    send("localhost",                    // smtp host name
      "your.address@your-company.com",       // from email address
      "You",                           // from display email
      "somebody@some-company.com",          // to email address
      "Subject test zh CN",            // subject
      "Content ˜4E02 from Text email", // body
      new Locale("zh", "CN")           // user locale
    );
  }
  public static void send(String smtp, String fromEmail, String fromDispName,
    String toEmail, String subject, String content, Locale locale
  )
  {
    // get the list of common email character sets
    final String[] charset = LocaleMapper.getIANACharSetFromLocale(LocaleMapper.
EMAIL_WINDOWS,
```

```
locale
      );
    // pick the first one for the email encoding
    final String contentType = "text/plain; charset=" + charset[0];
    try
    {
      Properties props = System.getProperties();
      props.put("mail.smtp.host", smtp);
      //  here, set username / password if necessary
      Session session = Session.getDefaultInstance(props, null);
      MimeMessage mimeMessage = new MimeMessage(session);
      mimeMessage.setFrom(new InternetAddress(fromEmail, fromDispName,
          charset[0]
        )
      );
      mimeMessage.setRecipients(Message.RecipientType.TO, toEmail);
      mimeMessage.setSubject(MimeUtility.encodeText(subject, charset[0], "Q"));
      // body
      mimeMessage.setContent(content, contentType);
      mimeMessage.setHeader("Content-Type", contentType);
      mimeMessage.setHeader("Content-Transfer-Encoding", "8bit");
      mimeMessage.setSentDate(new Date());
      Transport.send(mimeMessage);
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

The GDK Application Configuration File

The GDK application configuration file dictates the behavior and the properties of the GDK application framework and the application that is using it. It contains locale mapping tables and parameters for the configuration of the application. One configuration file is required for each application.

The `gdkapp.xml` application configuration file is an XML document. This file resides in the `./WEB-INF` directory of the J2EE environment of the application.

The following sections describe the contents and the properties of the application configuration file in detail:

- locale-charset-maps
- page-charset
- application-locales
- locale-determine-rule
- locale-parameter-name
- message-bundles
- url-rewrite-rule
- Example: GDK Application Configuration File

locale-charset-maps

This section enables applications to override the mapping from the language to the default character set provided by the GDK. This mapping is used when the `page-charset` is set to `AUTO-CHARSET`.

For example, for the `en` locale, the default GDK character set is `windows-1252`. However, if the application requires `ISO-8859-1`, this can be specified as follows:

```
<locale-charset-maps>
  <locale-charset>
    <locale>en</locale>
    <charset>ISO_8859-1</charset>
  </locale-charset>
</locale-charset-maps>
```

The locale name is comprised of the language code and the country code, and they should follow the ISO naming convention as defined in ISO 639 and ISO 3166, respectively. The character set name follows the IANA convention.

Optionally, the user-agent parameter can be specified in the mapping table to distinguish different clients as follows:

```
<locale-charset>
  <locale>en,de</locale>
  <user-agent>^Mozilla/4.0</user-agent>
  <charset>ISO-8859-1</charset>
</locale-charset>
```

The previous example shows that if the `user-agent` value in the HTTP header starts with `Mozilla/4.0` (which indicates an older version of Web clients) for English (`en`) and German (`de`) locales, then the GDK sets the character set to ISO-8859-1.

Multiple locales can be specified in a comma-delimited list.

See Also: "page-charset" on page 8-36

page-charset

This tag section defines the character set of the application pages. If this is explicitly set to a given character set, then all pages use this character set. The character set name must follow the IANA character set convention, for example:

```
<page-charset>UTF-8</page-charset>
```

However, if the `page-charset` is set to `AUTO-CHARSET`, then the character set is based on the default character set of the current user locale. The default character set is derived from the locale to character set mapping table specified in the application configuration file.

If the character set mapping table in the application configuration file is not available, then the character set is based on the default locale name to IANA character set mapping table in the GDK. Default mappings are derived from `OraLocaleInfo` class.

See Also:

- "locale-charset-maps" on page 8-36
- "Handling Non-ASCII Input and Output in the GDK Framework" on page 8-23

application-locales

This tag section defines a list of the locales supported by the application. For example:

```
<application-locales>
  <locale default="yes">en-US</locale>
  <locale>de</locale>
  <locale>zh-CN</locale>
</application-locales>
```

If the language component is specified with the * country code, then all locale names with this language code qualify. For example, if de-* (the language code for German) is defined as one of the application locales, then this supports de-AT (German- Austria), de (German-Germany), de-LU (German-Luxembourg), de-CH (German-Switzerland), and even irregular locale combination such as de-CN (German-China). However, the application can be restricted to support a predefined set of locales.

It is recommended to set one of the application locales as the default application locale (by specifying default="yes") so that it can be used as a fall back locale for customers who are connecting to the application with an unsupported locale.

locale-determine-rule

This section defines the order in which the preferred user locale is determined. The locale sources should be specified based on the scenario in the application. This section includes the following scenarios:

- Scenario 1: The GDK framework uses the accept language at all times.

  ```
  <locale-source>oracle.i18n.servlet.localesource.HTTPAcceptLanguage</locale-source>
  ```

- Scenario 2: By default, the GDK framework uses the accept language. After the user specifies the locale, the locale is used for further operations.

  ```
  <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>
  <locale-source>oracle.i18n.servlet.localesource.HTTPAcceptLanguage</locale-source>
  ```

- Scenario 3: By default, the GDK framework uses the accept language. After the user is authenticated, the GDK framework uses the database locale source. The database locale source is cached until the user logs out. After the user logs out, the accept language is used again.

  ```
  <db-locale-source
     data-source-name="jdbc/OracleCoreDS"
     locale-source-table="customer"
     user-column="customer_email"
     user-key="userid"
     language-column="nls_language"
     territory-column="nls_territory"
     timezone-column="timezone"
     >oracle.i18n.servlet.localesource.DBLocaleSource</db-locale-source>
  <locale-source>oracle.i18n.servlet.localesource.HttpAcceptLanguage</locale-source>
  ```

Note that Scenario 3 includes the predefined database locale source, DBLocaleSource. It enables the user profile information to be specified in the configuration file without writing a custom database locale source. In the example, the user profile table is called "customer". The columns are "customer_email", "nls_language", "nls_territory", and "timezone". They store the unique e-mail address, the Oracle name of the preferred language, the Oracle name of the preferred territory, and the time zone ID of a customer. The user-key is a mandatory attribute that specifies the attribute name used to pass the user ID from the application to the GDK framework.

- Scenario 4: The GDK framework uses the accept language in the first page. When the user inputs a locale, it is cached and used until the user logs into the application. After the user is authenticated, the GDK framework uses the database locale source. The database locale source is cached until the user logs out. After the user logs out, the accept language is used again or the user input is used if the user inputs a locale.

```
<locale-source>demo.DatabaseLocaleSource</locale-source>
<locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>
<locale-source>oracle.i18n.servlet.localesource.HttpAcceptLanguage</locale-source>
```

Note that Scenario 4 uses the custom database locale source. If the user profile schema is complex, such as user profile information separated into multiple tables, then the custom locale source should be provided by the application. Examples of custom locale sources can be found in the `$ORACLE_HOME/nls/gdk/demo` directory.

locale-parameter-name

This tag defines the name of the locale parameters that are used in the user input so that the current user locale can be passed between requests.

Table 8–3 shows the parameters used in the GDK framework.

Table 8–3 Locale Parameters Used in the GDK Framework

Default Parameter Name	Value
`locale`	ISO locale where ISO 639 language code and ISO 3166 country code are connected with an underscore (_).or a hyphen (-). For example, `zh_CN` for Simplified Chinese used in China
`language`	Oracle language name. For example, `AMERICAN` for American English
`territory`	Oracle territory name. For example, `SPAIN`
`timezone`	Time zone name. For example, `American/Los_Angeles`
`iso-currency`	ISO 4217 currency code. For example, `EUR` for the euro
`date-format`	Date format pattern mask. For example, `DD_MON_RRRR`
`long-date-format`	Long date format pattern mask. For example, `DAY-YYY-MM-DD`
`date-time-format`	Date and time format pattern mask. For example, `DD-MON-RRRR HH24:MI:SS`
`long-date-time-format`	Long date and time format pattern mask. For example, `DAY YYYY-MM-DD HH12:MI:SS AM`
`time-format`	Time format pattern mask. For example, `HH:MI:SS`
`number-format`	Number format. For example, `9G99G990D00`
`currency-format`	Currency format. For example, `L9G99G990D00`
`linguistic-sorting`	Linguistic sort order name. For example, `JAPANESE_M` for Japanese multilingual sort
`charset`	Character set. For example, `WE8ISO8859P15`
`writing-direction`	Writing direction string. For example, `LTR` for left-to-right writing direction or `RTL` for right-to-left writing direction
`command`	GDK command. For example, `store` for the update operation

The parameter names are used in either the parameter in the HTML form or in the URL.

message-bundles

This tag defines the base class names of the resource bundles used in the application. The mapping is used in the `Localizer.getMessage` method for locating translated text in the resource bundles.

```
<message-bundles>
  <resource-bundle>Messages</resource-bundle>
  <resource-bundle name="newresource">NewMessages</resource-bundle>
</message-bundles>
```

If the name attribute is not specified or if it is specified as `name="default"` to the `<resource-bundle>` tag, then the corresponding resource bundle is used as the default message bundle. To support more than one resource bundle in an application, resource bundle names must be assigned to the nondefault resource bundles. The nondefault bundle names must be passed as a parameter of the `getMessage` method.

For example:

```
Localizer loc = ServletHelper.getLocalizerInstance(request);
String translatedMessage = loc.getMessage("Hello");
String translatedMessage2 = loc.getMessage("World", "newresource");
```

url-rewrite-rule

This tag is used to control the behavior of the URL rewrite operations. The rewriting rule is a regular expression.

```
<url-rewrite-rule fallback="no">
  <pattern>(.*)/([^/]+)$</pattern>
  <result>$1/$L/$2</result>
</url-rewrite-rule>
```

See Also: "Managing Localized Content in the GDK" on page 8-25

If the localized content for the requested locale is not available, then it is possible for the GDK framework to trigger the locale fallback mechanism by mapping it to the closest translation locale. By default, the fallback option is turned off. This can be turned on by specifying `fallback="yes"`.

For example, suppose an application supports only the following translations: `en`, `de`, and `ja`, and `en` is the default locale of the application. If the current application locale is `de-US`, then it falls back to `de`. If the user selects `zh-TW` as its application locale, then it falls back to `en`.

A fallback mechanism is often necessary if the number of supported application locales is greater than the number of the translation locales. This usually happens if multiple locales share one translation. One example is Spanish. The application may need to support multiple Spanish-speaking countries and not just Spain, with one set of translation files.

Multiple URL rewrite rules can be specified by assigning the name attribute to nondefault URL rewrite rules. To use the nondefault URL rewrite rules, the name must be passed as a parameter of the rewrite URL method. For example:

```
<img src="<%=ServletHelper.rewriteURL("images/welcome.gif", request) %>">
<img src="<%=ServletHelper.rewriteURL("US.gif", "flag", request) %>">
```

The first rule changes the `"images/welcome.gif"` URL to the localized welcome image file. The second rule named `"flag"` changes the `"US.gif"` URL to the user's country flag image file. The rule definition should be as follows:

```
<url-rewrite-rule fallback="yes">
  <pattern>(.*)/([^/]+)$</pattern>
  <result>$1/$L/$2</result>
</url-rewrite-rule>
<url-rewrite-rule name="flag">
  <pattern>US.gif/pattern>
  <result>$C.gif</result>
</url-rewrite-rule>
```

Example: GDK Application Configuration File

This section contains an example of an application configuration file with the following application properties:

- The application supports the following locales: Arabic (ar), Greek (`el`), English (`en`), German (`de`), French (`fr`), Japanese (`ja`) and Simplified Chinese for China (`zh-CN`).
- English is the default application locale.
- The page character set for the `ja` locale is always UTF-8.
- The page character set for the `en` and `de` locales when using an Internet Explorer client is `windows-1252`.
- The page character set for the `en`, `de`, and `fr` locales on other web browser clients is `iso-8859-1`.
- The page character sets for all other locales are the default character set for the locale.
- The user locale is determined by the following order: user input locale and then `Accept-Language`.
- The localized contents are stored in their appropriate language subfolders. The folder names are derived from the ISO 639 language code. The folders are located in the root directory of the application. For example, the Japanese file for `/shop/welcome.jpg` is stored in `/ja/shop/welcome.jpg`.

```
<?xml version="1.0" encoding="utf-8"?>
<gdkapp
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="gdkapp.xsd">
  <!-- Language to Character set mapping -->
  <locale-charset-maps>
    <locale-charset>
      <locale>ja</locale>
      <charset>UTF-8</charset>
    </locale-charset>
    <locale-charset>
      <locale>en,de</locale>
      <user-agent>^Mozilla\/[0-9\. ]+\(compatible; MSIE [^;]+; \)</user-agent>
      <charset>WINDOWS-1252</charset>
    </locale-charset>
    <locale-charset>
      <locale>en,de,fr</locale>
      <charset>ISO-8859-1</charset>
    </locale-charset>
  </locale-charset-maps>

  <!-- Application Configurations -->
  <page-charset>AUTO-CHARSET</page-charset>
  <application-locales>
    <locale>ar</locale>
    <locale>de</locale>
```

```
    <locale>fr</locale>
    <locale>ja</locale>
    <locale>el</locale>
    <locale default="yes">en</locale>
    <locale>zh-CN</locale>
  </application-locales>
  <locale-determine-rule>
    <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>
    <locale-source>oracle.i18n.servlet.localesource.HttpAcceptLanguage</locale-source>
  </locale-determine-rule>
  <!-- URL rewriting rule -->
  <url-rewrite-rule fallback="no">
    <pattern>(.*)/([^/]+)$</pattern>
    <result>/$L/$1/$2</result>
  </url-rewrite-rule>
</gdkapp>
```

GDK for Java Supplied Packages and Classes

Oracle Globalization Services for Java contains the following packages:

- oracle.i18n.lcsd
- oracle.i18n.net
- oracle.i18n.servlet
- oracle.i18n.text
- oracle.i18n.util

See Also: *Oracle Globalization Development Kit Java API Reference*

oracle.i18n.lcsd

Package `oracle.i18n.lcsd` provides classes to automatically detect and recognize language and character set based on text input. It supports the detection of both plain text and HTML files. Language is based on ISO; encoding is based on IANA or Oracle character sets. It includes the following classes:

- `LCSDetector`: Contains methods to automatically detect and recognize language and character set based on text input.
- `LCSDResultSet`: The `LCSDResultSet` class is for storing the result generated by `LCSDetector`. Methods in this class can be used to retrieve specific information from the result.
- `LCSDetectionInputStream`: Transparently detects the language and encoding for the stream object.
- `LCSDetectionReader`: Transparently detects the language and encoding and converts the input data to Unicode.
- `LCSDetectionHTMLInputStream`: Extends the `LCSDetectionInputStream` class to support the language and encoding detection for input in HTML format.
- `LCSDetectionHTMLReader`: Extends the `LCSDetectionReader` class to support the language and encoding detection for input in HTML format.

LCSScan

The Language and Character Set File Scanner (Java Version) is a statistically-based utility for determining the language and character set for unknown file text. Its

functionality and usage are similar to the Language and Character Set File Scanner of the "C" Version.

> **See Also:** "The Language and Character Set File Scanner" on page 11-8

Syntax of the LCSScan Command

```
Usage: java LCSScan <options>
Example: java LCSScan FILE=test.txt RESULTS=3 SIZE=10000

Keyword   Description                                              (Default)
-------   -------------                                            --------
RESULTS   number of language and character set pairs to return 1..3  1
SIZE      sampling size of the file in bytes                       8192
FORMAT    file format TEXT or HTML                                 TEXT
RATIO     show result ratio YES or NO                              NO
FILE      name of input file
HELP      show help screen                                         this screen
```

Examples of Using LCSScan Make sure that the `orai18n-lcsd.jar` and `orai18n-mapping.jar` files are in the `CLASSPATH`.

Example 8–10 Specifying the File Name in the LCSScan Command

```
java oracle/i18n/lcsd/LCSScan FILE=example.txt
```

In this example, 8192 bytes of `example.txt` file is scanned. One language and character set pair will be returned.

Example 8–11 Specifying the File Name and Sampling Size in the LCSScan Command

```
java oracle/i18n/lcsd/LCSScan FILE=example.txt SIZE=4096
```

In this example, 4096 bytes of `example.txt` file is scanned. One language and character set pair will be returned.

Example 8–12 Specifying the File Name and Number of Language and Character Set Pairs in the LCSScan Command

```
java oracle/i18n/lcsd/LCSScan FILE=example.txt RESULTS=3
```

In this example, 8192 bytes of `example.txt` file is scanned. Three language and character set pairs will be returned.

Example 8–13 Specifying the File Name and Show Result Ratio in the LCSScan Command

```
java oracle/i18n/lcsd/LCSScan FILE=example.txt RATIO=YES
```

In this example, 8192 bytes of `example.txt` file is scanned. One language and character set pair will be returned with the result ratio.

Example 8–14 Specifying the File Name and Format as HTML

```
java oracle/i18n/lcsd/LCSScan FILE=example.html FORMAT=html
```

In this example, 8192 bytes of `example.html` file is scanned. The scan will strip HTML tags before the scan, thus results are more accurate. One language and character set pair will be returned.

oracle.i18n.net

Package `oracle.i18n.net` provides Internet-related data conversions for globalization. It includes the following classes:

- `CharEntityReference`: A utility class to escape or unescape a string into character reference or entity reference form.
- `CharEntityReference.Form`: A form parameter class that specifies the escaped form.

oracle.i18n.servlet

Package `oracle.i18n.Servlet` enables JSP and JavaServlet to have automatic locale support and also returns the localized contents to the application. It includes the following classes:

- `ApplicationContext`: An application context class that governs application scope operation in the framework.
- `Localizer`: An all-in-one object class that enables access to the most commonly used globalization information.
- `ServletHelper`: A delegate class that bridges between Java servlets and globalization objects.

oracle.i18n.text

Package `oracle.i18n.text` provides general text data globalization support. It includes the following classes:

- `OraCollationKey`: A class which represents a `String` under certain rules of a specific `OraCollator` object.
- `OraCollator`: A class to perform locale-sensitive string comparison, including linguistic collation and binary sorting.
- `OraDateFormat`: An abstract class to do formatting and parsing between datetime and string locale. It supports Oracle datetime formatting behavior.
- `OraDecimalFormat`: A concrete class to do formatting and parsing between number and string locale. It supports Oracle number formatting behavior.
- `OraDecimalFormatSymbol`: A class to maintain Oracle format symbols used by Oracle number and currency formatting.
- `OraNumberFormat`: An abstract class to do formatting and parsing between number and string locale. It supports Oracle number formatting behavior.
- `OraSimpleDateFormat`: A concrete class to do formatting and parsing between datetime and string locale. It supports Oracle datetime formatting behavior.

oracle.i18n.util

Package `oracle.i18n.util` provides general utilities for globalization support. It includes the following classes:

- `LocaleMapper`: Provides mappings between Oracle locale elements and equivalent locale elements in other vendors and standards.
- `OraDisplayLocaleInfo`: A translation utility class that provides the translations of locale and attributes.

- `OraLocaleInfo`: An Oracle locale class that includes the language, territory, and collator objects.
- `OraSQLUtil`: An Oracle SQL Utility class that includes some useful methods of dealing with SQL.

GDK for PL/SQL Supplied Packages

The GDK for PL/SQL includes the following PL/SQL packages:

- `UTL_I18N`
- `UTL_LMS`

`UTL_I18N` is a set of PL/SQL services that help developers to build globalized applications. The `UTL_I18N` PL/SQL package provides the following functions:

- String conversion functions for various data types
- Escape and unescape sequences for predefined characters and multibyte characters used by HTML and XML documents
- Functions that map between Oracle, Internet Assigned Numbers Authority (IANA), ISO, and e-mail application character sets, languages, and territories
- A function that returns the Oracle character set name from an Oracle language name
- A function that performs script transliteration
- Functions that return the ISO currency code, local time zones, and local languages supported for a given territory
- Functions that return the most commonly used linguistic sort, a listing of all applicable linguistic sorts, and the local territories supported for a given language
- Functions that map between Oracle full and short language names
- A function that returns the language translation of a given language and territory name
- A function that returns a listing of the most commonly used time zones
- A function that returns the maximum number of bytes for a character of an Oracle character set

`UTL_LMS` retrieves and formats error messages in different languages.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

GDK Error Messages

GDK-03001 Invalid or unsupported sorting rule

Cause: An invalid or unsupported sorting rule name was specified.

Action: Choose a valid sorting rule name and check the Globalization Support Guide for the list of sorting rule names.

GDK-03002 The functional-driven sort is not supported.

Cause: A functional-driven sorting rule name was specified.

Action: Choose a valid sorting rule name and check the Globalization Support Guide for the list of sorting rule names.

GDK-03003 The linguistic data file is missing.

Cause: A valid sorting rule was specified, but the associated data file was not found.

Action: Make sure the GDK jar files are correctly installed in the Java application.

GDK-03005 Binary sort is not available for the specified character set .

Cause: Binary sorting for the specified character set is not supported.

Action: Check the Globalization Support Guide for a character set that supports binary sort.

GDK-03006 The comparison strength level setting is invalid.

Cause: An invalid comparison strength level was specified.

Action: Choose a valid comparison strength level from the list -- PRIMARY, SECONDARY or TERTIARY.

GDK-03007 The composition level setting is invalid.

Cause: An invalid composition level setting was specified.

Action: Choose a valid composition level from the list -- NO_COMPOSITION or CANONICAL_COMPOSITION.

GDK-04001 Cannot map Oracle character to Unicode

Cause: The program attempted to use a character in the Oracle character set that cannot be mapped to Unicode.

Action: Write a separate exception handler for the invalid character, or call the withReplacement method so that the invalid character can be replaced with a valid replacement character.

GDK-04002 Cannot map Unicode to Oracle character

Cause: The program attempted to use an Unicode character that cannot be mapped to a character in the Oracle character set.

Action: Write a separate exception handler for the invalid character, or call the withReplacement method so that the invalid character can be replaced with a valid replacement character.

GDK-05000 A literal in the date format is too large.

Cause: The specified string literal in the date format was too long.

Action: Use a shorter string literal in the date format.

GDK-05001 The date format is too long for internal buffer.

Cause: The date format pattern was too long.

Action: Use a shorter date format pattern.

GDK-05002 The Julian date is out of range.

Cause: An illegal date range was specified.

Action: Make sure that date is in the specified range 0 - 3439760.

GDK-05003 Failure in retrieving date/time

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05010 Duplicate format code found

Cause: The same format code was used more than once in the format pattern.

Action: Remove the redundant format code.

GDK-05011 The Julian date precludes the use of the day of the year.
Cause: Both the Julian date and the day of the year were specified.
Action: Remove either the Julian date or the day of the year.

GDK-05012 The year may only be specified once.
Cause: The year format code appeared more than once.
Action: Remove the redundant year format code.

GDK-05013 The hour may only be specified once.
Cause: The hour format code appeared more than once.
Action: Remove the redundant hour format code.

GDK-05014 The AM/PM conflicts with the use of A.M./P.M.
Cause: AM/PM was specified along with A.M./P.M.
Action: Use either AM/PM or A.M./P.M; do not use both.

GDK-05015 The BC/AD conflicts with the use of B.C./A.D.
Cause: BC/AD was specified along with B.C./A.D.
Action: Use either BC/AD or B.C./A.D.; do not use both.

GDK-05016 Duplicate month found
Cause: The month format code appeared more than once.
Action: Remove the redundant month format code.

GDK-05017 The day of the week may only be specified once.
Cause: The day of the week format code appeared more than once.
Action: Remove the redundant day of the week format code.

GDK-05018 The HH24 precludes the use of meridian indicator.
Cause: HH24 was specified along with the meridian indicator.
Action: Use either the HH24 or the HH12 with the meridian indicator.

GDK-05019 The signed year precludes the use of BC/AD.
Cause: The signed year was specified along with BC/AD.
Action: Use either the signed year or the unsigned year with BC/AD.

GDK-05020 A format code cannot appear in a date input format.
Cause: A format code appeared in a date input format.
Action: Remove the format code.

GDK-05021 Date format not recognized
Cause: An unsupported format code was specified.
Action: Correct the format code.

GDK-05022 The era format code is not valid with this calendar.
Cause: An invalid era format code was specified for the calendar.
Action: Remove the era format code or use anther calendar that supports the era.

GDK-05030 The date format pattern ends before converting entire input string.

Cause: An incomplete date format pattern was specified.

Action: Rewrite the format pattern to cover the entire input string.

GDK-05031 The year conflicts with the Julian date.

Cause: An incompatible year was specified for the Julian date.

Action: Make sure that the Julian date and the year are not in conflict.

GDK-05032 The day of the year conflicts with the Julian date.

Cause: An incompatible day of year was specified for the Julian date.

Action: Make sure that the Julian date and the day of the year are not in conflict.

GDK-05033 The month conflicts with the Julian date.

Cause: An incompatible month was specified for the Julian date.

Action: Make sure that the Julian date and the month are not in conflict.

GDK-05034 The day of the month conflicts with the Julian date.

Cause: An incompatible day of the month was specified for the Julian date.

Action: Make sure that the Julian date and the day of the month are not in conflict.

GDK-05035 The day of the week conflicts with the Julian date.

Cause: An incompatible day of the week was specified for the Julian date.

Action: Make sure that the Julian date and the day of week are not in conflict.

GDK-05036 The hour conflicts with the seconds in the day.

Cause: The specified hour and the seconds in the day were not compatible.

Action: Make sure the hour and the seconds in the day are not in conflict.

GDK-05037 The minutes of the hour conflicts with the seconds in the day.

Cause: The specified minutes of the hour and the seconds in the day were not compatible.

Action: Make sure the minutes of the hour and the seconds in the day are not in conflict.

GDK-05038 The seconds of the minute conflicts with the seconds in the day.

Cause: The specified seconds of the minute and the seconds in the day were not compatible.

Action: Make sure the seconds of the minute and the seconds in the day are not in conflict.

GDK-05039 Date not valid for the month specified

Cause: An illegal date for the month was specified.

Action: Check the date range for the month.

GDK-05040 Input value not long enough for the date format

Cause: Too many format codes were specified.

Action: Remove unused format codes or specify a longer value.

GDK-05041 A full year must be between -4713 and +9999, and not be 0.

Cause: An illegal year was specified.

Action: Specify the year in the specified range.

GDK-05042 A quarter must be between 1 and 4.

Cause: Cause: An illegal quarter was specified.

Action: Action: Make sure that the quarter is in the specified range.

GDK-05043 Not a valid month

Cause: An illegal month was specified.

Action: Make sure that the month is between 1 and 12 or has a valid month name.

GDK-05044 The week of the year must be between 1 and 52.

Cause: An illegal week of the year was specified.

Action: Make sure that the week of the year is in the specified range.

GDK-05045 The week of the month must be between 1 and 5.

Cause: An illegal week of the month was specified.

Action: Make sure that the week of the month is in the specified range.

GDK-05046 Not a valid day of the week

Cause: An illegal day of the week was specified.

Action: Make sure that the day of the week is between 1 and 7 or has a valid day name.

GDK-05047 A day of the month must be between 1 and the last day of the month.

Cause: An illegal day of the month was specified.

Action: Make sure that the day of the month is in the specified range.

GDK-05048 A day of year must be between 1 and 365 (366 for leap year).

Cause: An illegal day of the year was specified.

Action: Make sure that the day of the year is in the specified range.

GDK-05049 An hour must be between 1 and 12.

Cause: An illegal hour was specified.

Action: Make sure that the hour is in the specified range.

GDK-05050 An hour must be between 0 and 23.

Cause: An illegal hour was specified.

Action: Make sure that the hour is in the specified range.

GDK-05051 A minute must be between 0 and 59.

Cause: Cause: An illegal minute was specified.

Action: Action: Make sure the minute is in the specified range.

GDK-05052 A second must be between 0 and 59.

Cause: An illegal second was specified.

Action: Make sure the second is in the specified range.

GDK-05053 A second in the day must be between 0 and 86399.

Cause: An illegal second in the day was specified.

Action: Make sure second in the day is in the specified range.

GDK-05054 The Julian date must be between 1 and 5373484.

Cause: An illegal Julian date was specified.

Action: Make sure that the Julian date is in the specified range.

GDK-05055 Missing AM/A.M. or PM/P.M.

Cause: Neither AM/A.M. nor PM/P.M. was specified in the format pattern.

Action: Specify either AM/A.M. or PM/P.M.

GDK-05056 Missing BC/B.C. or AD/A.D.

Cause: Neither BC/B.C. nor AD/A.D. was specified in the format pattern.

Action: Specify either BC/B.C. or AD/A.D.

GDK-05057 Not a valid time zone

Cause: An illegal time zone was specified.

Action: Specify a valid time zone.

GDK-05058 Non-numeric character found

Cause: A non-numeric character was found where a numeric character was expected.

Action: Make sure that the character is a numeric character.

GDK-05059 Non-alphabetic character found

Cause: A non-alphabetic character was found where an alphabetic was expected.

Action: Make sure that the character is an alphabetic character.

GDK-05060 The week of the year must be between 1 and 53.

Cause: An illegal week of the year was specified.

Action: Make sure that the week of the year is in the specified range.

GDK-05061 The literal does not match the format string.

Cause: The string literals in the input were not the same length as the literals in the format pattern (with the exception of the leading whitespace).

Action: Correct the format pattern to match the literal. If the "FX" modifier has been toggled on, the literal must match exactly, with no extra whitespace.

GDK-05062 The numeric value does not match the length of the format item.

Cause: The numeric value did not match the length of the format item.

Action: Correct the input date or turn off the FX or FM format modifier. When the FX and FM format codes are specified for an input date, then the number of digits must be exactly the number specified by the format code. For example, 9 will not match the format code DD but 09 will.

GDK-05063 The year is not supported for the current calendar.

Cause: An unsupported year for the current calendar was specified.

Action: Check the Globalization Support Guide to find out what years are supported for the current calendar.

GDK-05064 The date is out of range for the calendar.

Cause: The specified date was out of range for the calendar.

Action: Specify a date that is legal for the calendar.

GDK-05065 Invalid era

Cause: An illegal era was specified.

Action: Make sure that the era is valid.

GDK-05066 The datetime class is invalid.

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05067 The interval is invalid.

Cause: An invalid interval was specified.

Action: Specify a valid interval.

GDK-05068 The leading precision of the interval is too small.

Cause: The specified leading precision of the interval was too small to store the interval.

Action: Increase the leading precision of the interval or specify an interval with a smaller leading precision.

GDK-05069 Reserved for future use

Cause: Reserved.

Action: Reserved.

GDK-05070 The specified intervals and datetimes were not mutually comparable.

Cause: The specified intervals and datetimes were not mutually comparable.

Action: Specify a pair of intervals or datetimes that are mutually comparable.

GDK-05071 The number of seconds must be less than 60.

Cause: The specified number of seconds was greater than 59.

Action: Specify a value for the seconds to 59 or smaller.

GDK-05072 Reserved for future use

Cause: Reserved.

Action: Reserved.

GDK-05073 The leading precision of the interval was too small.

Cause: The specified leading precision of the interval was too small to store the interval.

Action: Increase the leading precision of the interval or specify an interval with a smaller leading precision.

GDK-05074 An invalid time zone hour was specified.

Cause: The hour in the time zone must be between -12 and 13.

Action: Specify a time zone hour between -12 and 13.

GDK-05075 An invalid time zone minute was specified.

Cause: The minute in the time zone must be between 0 and 59.

Action: Specify a time zone minute between 0 and 59.

GDK-05076 An invalid year was specified.

Cause: A year must be at least -4713.

Action: Specify a year that is greater than or equal to -4713.

GDK-05077 The string is too long for the internal buffer.

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05078 The specified field was not found in the datetime or interval.

Cause: The specified field was not found in the datetime or interval.

Action: Make sure that the specified field is in the datetime or interval.

GDK-05079 An invalid hh25 field was specified.

Cause: The hh25 field must be between 0 and 24.

Action: Specify an hh25 field between 0 and 24.

GDK-05080 An invalid fractional second was specified.

Cause: The fractional second must be between 0 and 999999999.

Action: Specify a value for fractional second between 0 and 999999999.

GDK-05081 An invalid time zone region ID was specified.

Cause: The time zone region ID specified was invalid.

Action: Contact Oracle Support Services.

GDK-05082 Time zone region name not found

Cause: The specified region name cannot be found.

Action: Contact Oracle Support Services.

GDK-05083 Reserved for future use

Cause: Reserved.

Action: Reserved.

GDK-05084 Internal formatting error

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05085 Invalid object type

Cause: An illegal object type was specified.

Action: Use a supported object type.

GDK-05086 Invalid date format style

Cause: An illegal format style was specified.

Action: Choose a valid format style.

GDK-05087 A null format pattern was specified.

Cause: The format pattern cannot be null.

Action: Provide a valid format pattern.

GDK-05088 Invalid number format model

Cause: An illegal number format code was specified.

Action: Correct the number format code.

GDK-05089 Invalid number

Cause: An invalid number was specified.

Action: Correct the input.

GDK-05090 Reserved for future use

Cause: Reserved.

Action: Reserved.

GDK-0509 Datetime/interval internal error

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05098 Too many precision specifiers

Cause: Extra data was found in the date format pattern while the program attempted to truncate or round dates.

Action: Check the syntax of the date format pattern.

GDK-05099 Bad precision specifier

Cause: An illegal precision specifier was specified.

Action: Use a valid precision specifier.

GDK-05200 Missing WE8ISO8859P1 data file

Cause: The character set data file for WE8ISO8859P1 was not installed.

Action: Make sure the GDK jar files are installed properly in the Java application.

GDK-05201 Failed to convert to a hexadecimal value

Cause: An invalid hexadecimal string was included in the HTML/XML data.

Action: Make sure the string includes the hexadecimal character in the form of &x[0-9A-Fa-f]+;.

GDK-05202 Failed to convert to a decimal value

Cause: An invalid decimal string was found in the HTML/XML data.

Action: Make sure the string includes the decimal character in the form of &[0-9]+;.

GDK-05203 Unregistered character entity

Cause: An invalid character entity was found in the HTML/XML data.

Action: Use a valid character entity value in HTML/XML data. See HTML/XML standards for the registered character entities.

GDK-05204 Invalid Quoted-Printable value

Cause: An invalid Quoted-Printable data was found in the data.

Action: Make sure the input data has been encoded in the proper Quoted-Printable form.

GDK-05205 Invalid MIME header format

Cause: An invalid MIME header format was specified.

Action: Check RFC 2047 for the MIME header format. Make sure the input data conforms to the format.

GDK-05206 Invalid numeric string

Cause: An invalid character in the form of %FF was found when a URL was being decoded.

Action: Make sure the input URL string is valid and has been encoded correctly; %FF needs to be a valid hex number.

GDK-05207 Invalid class of the object, key, in the user-defined locale to charset mapping"

Cause: The class of key object in the user-defined locale to character set mapping table was not java.util.Locale.

Action: When you construct the Map object for the user-defined locale to character set mapping table, specify java.util.Locale for the key object.

GDK-05208 Invalid class of the object, value, in the user-defined locale to charset mapping

Cause: The class of value object in the user-defined locale to character set mapping table was not java.lang.String.

Action: When you construct the Map object for the user-defined locale to character set mapping table, specify java.lang.String for the value object.

GDK-05209 Invalid rewrite rule

Cause: An invalid regular expression was specified for the match pattern in the rewrite rule.

Action: Make sure the match pattern for the rewriting rule uses a valid regular expression.

GDK-05210 Invalid character set

Cause: An invalid character set name was specified.

Action: Specify a valid character set name.

GDK-0521 Default locale not defined as a supported locale

Cause: The default application locale was not included in the supported locale list.

Action: Include the default application locale in the supported locale list or change the default locale to the one that is in the list of the supported locales.

GDK-05212 The rewriting rule must be a String array with three elements.

Cause: The rewriting rule parameter was not a String array with three elements.

Action: Make sure the rewriting rule parameter is a String array with three elements. The first element represents the match pattern in the regular expression, the second element represents the result pattern in the form specified in the JavaDoc of ServletHelper.rewriteURL, and the third element represents the Boolean value "True" or "False" that specifies whether the locale fallback operation is performed or not.

GDK-05213 Invalid type for the class of the object, key, in the user-defined parameter name mapping

Cause: The class of key object in the user-defined parameter name mapping table was not java.lang.String.

Action: When you construct the Map object for the user-defined parameter name mapping table, specify java.lang.String for the key object.

GDK-05214 The class of the object, value, in the user-defined parameter name mapping, must be of type \"java.lang.String\".

Cause: The class of value object in the user-defined parameter name mapping table was not java.lang.String.

Action: When you construct the Map object for the user-defined parameter name mapping table, specify java.lang.String for the value object.

GDK-05215 Parameter name must be in the form [a-z][a-z0-9]*.

Cause: An invalid character was included in the parameter name.

Action: Make sure the parameter name is in the form of [a-z][a-z0-9]*.

GDK-05216 The attribute \"var\" must be specified if the attribute \"scope\" is set.

Cause: Despite the attribute "scope" being set in the tag, the attribute "var" was not specified.

Action: Specify the attribute "var" for the name of variable.

GDK-05217 The \"param\" tag must be nested inside a \"message\" tag.

Cause: The "param" tag was not nested inside a "message" tag.

Action: Make sure the tag "param" is inside the tag "message".

GDK-05218 Invalid \"scope\" attribute is specified.

Cause: An invalid "scope" value was specified.

Action: Specify a valid scope as either "application," "session," "request," or "page".

GDK-05219 Invalid date format style

Cause: The specified date format style was invalid.

Action: Specify a valid date format style as either "default," "short," or "long"

GDK-05220 No corresponding Oracle character set exists for the IANA character set.

Cause: An unsupported IANA character set name was specified.

Action: Specify the IANA character set that has a corresponding Oracle character set.

GDK-05221 Invalid parameter name

Cause: An invalid parameter name was specified in the user-defined parameter mapping table.

Action: Make sure the specified parameter name is supported. To get the list of supported parameter names, call LocaleSource.Parameter.toArray.

GDK-05222 Invalid type for the class of the object, key, in the user-defined message bundle mapping.

Cause: The class of key object in the user-defined message bundle mapping table was not "java.lang.String."

Action: When you construct the Map object for the user-defined message bundle mapping table, specify java.lang.String for the key object.

GDK-05223 Invalid type for the class of the object, value, in the user-defined message bundle mapping

Cause: The class of value object in the user-defined message bundle mapping table was not "java.lang.String."

Action: When you construct the Map object for the user-defined message bundle mapping table, specify java.lang.String for the value object.

GDK-05224 Invalid locale string

Cause: An invalid character was included in the specified ISO locale names in the GDK application configuration file.

Action: Make sure the ISO locale names include only valid characters. A typical name format is an ISO 639 language followed by an ISO 3166 country connected by a dash character; for example, "en-US" is used to specify the locale for American English in the United States.

GDK-06001 LCSDetector profile not available

Cause: The specified profile was not found.

Action: Make sure the GDK jar files are installed properly in the Java application.

GDK-06002 Invalid IANA character set name or no corresponding Oracle name found

Cause: The IANA character set specified was either invalid or did not have a corresponding Oracle character set.

Action: Check that the IANA character is valid and make sure that it has a corresponding Oracle character set.

GDK-06003 Invalid ISO language name or no corresponding Oracle name found

Cause: The ISO language specified was either invalid or did not have a corresponding Oracle language.

Action: Check to see that the ISO language specified is valid and has a corresponding Oracle language.

GDK-06004 A character set filter and a language filter cannot be set at the same time.

Cause: A character set filter and a language filter were set at the same time in a LCSDetector object.

Action: Set only one of the two -- character set or language.

GDK-06005 Reset is necessary before LCSDetector can work with a different data source.

Cause: The reset method was not invoked before a different type of data source was used for a LCSDetector object.

Action: Call LCSDetector.reset to reset the detector before switching to detect other types of data source.

ORA-17154 Cannot map Oracle character to Unicode

Cause: The Oracle character was either invalid or incomplete and could not be mapped to an Unicode value.

Action: Write a separate exception handler for the invalid character, or call the withReplacement method so that the invalid character can be replaced with a valid replacement character.

ORA-17155 Cannot map Unicode to Oracle character

Cause: The Unicode character did not have a counterpart in the Oracle character set.

Action: Write a separate exception handler for the invalid character, or call the withReplacement method so that the invalid character can be replaced with a valid replacement character.

9

SQL and PL/SQL Programming in a Global Environment

This chapter contains information useful for SQL programming in a globalization support environment. This chapter 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

NLS parameters can be specified for all SQL functions whose behavior depends on globalization support conventions. 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-2005';
```

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-2005','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.

> **Note:** Only SQL statements that must be independent of the session NLS parameter values should explicitly specify optional NLS parameters in locale-dependent SQL functions. Using session default values for NLS parameters in SQL functions usually results in better performance.

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_DUAL_CURRENCY
NLS_CALENDAR
NLS_SORT
```

Table 9–1 shows which NLS parameters are valid for specific SQL functions.

Table 9–1 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 9–1 (Cont.) SQL Functions and Their Valid NLS Parameters

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_CHAR (SYSDATE, 'DD/MON/YYYY',
    'nls_date_language=''Traditional Chinese'' ')

TO_NUMBER ('13.000,00', '99G999D99',
   'nls_numeric_characters = '',.''')

TO_CHAR (salary, '9G999D99L', 'nls_numeric_characters = '',.''
   nls_currency = ''EUR''')

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 5-13
- "Special Lowercase Letters" on page 5-13

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 and NLS_TERRITORY_FORMAT are not accepted as parameters because they 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 and NLS_TERRITORY_FORMAT are not valid NLS parameters for the TO_CHAR or TO_DATE functions. If you specify NLS_DATE_FORMAT or NLS_TERRITORY_FORMAT in the TO_CHAR or TO_DATE function, then an error is returned.

NLS_LANGUAGE can interfere with the session value of NLS_DATE_LANGUAGE. If you specify NLS_LANGUAGE in the TO_CHAR function, for example, then its value is ignored if it differs from the session value of NLS_DATE_LANGUAGE.

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 2-13

The syntax for CONVERT is as follows:

```
CONVERT(char, dest_char_set[, source_char_set])
```

char is the value to be converted. *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.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for more information about the CONVERT function
> - "Character Set Conversion Support" on page A-16 for character set encodings that are used only for the CONVERT function

SQL Functions for Different Length Semantics

Oracle 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. 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 9–2 summarizes the calculation methods.

Table 9–2 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 data type. For example, AL32UTF8 characters are calculated in UCS-4 characters. UTF8 and AL16UTF16 characters are calculated in UCS-2 characters. A supplementary character is counted as one character in AL32UTF8 and as two characters in UTF8 and AL16UTF16. Because VARCHAR and NVARCHAR2 may use different character sets, SUBSTR may give different results for different data types 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 characters, 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 characters.
SUBSTR4	Calculates the length of a string in UCS-4 characters. Characters are represented in UCS-4 or 32-bit Unicode values. Supplementary characters are counted as one character.
SUBSTRC	Calculates the length of a string in Unicode composed characters. Supplementary characters and composed 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:** *Oracle Database SQL Language 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 9–3.

Table 9–3 LIKE Conditions

Function	Description
LIKE2	Use when characters are represented in UCS-2 semantics. A supplementary character is considered as two characters.
LIKE4	Use when characters are represented in UCS-4 semantics. A supplementary character is considered as one character.
LIKEC	Use when characters are represented in Unicode complete character semantics. A composed character is treated as one character.

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 declaration length of a column in number of characters, given the byte length of the 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: *Oracle Database SQL Language 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 database character set. If the value NCHAR_CS is entered for *text*, then the function returns the ID of the database'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 of a column in number of characters, given the byte length of the 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 9–1 specifies a German sort with the NLS_SORT session parameter.

Example 9–1 Specifying a German Sort with the NLS_SORT Session Parameter

```
ALTER SESSION SET NLS_SORT = GERMAN;
SELECT * FROM table1
     ORDER BY column1;
```

Example 9–2 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 when NLS_COMP is set to BINARY, but this can be overridden by using the NLSSORT function in the WHERE clause.

Example 9–3 Making a Linguistic Comparison with the WHERE Clause

```
ALTER SESSION SET NLS_COMP = BINARY;
SELECT * FROM table1
WHERE NLSSORT(column1, 'NLS_SORT=FRENCH')>
      NLSSORT(column2, 'NLS_SORT=FRENCH');
```

Setting the NLS_COMP session parameter to LINGUISTIC 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, then 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 3-31

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: "Using Linguistic Collation" on page 5-3

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
- Loading External BFILE Data into LOB Columns

See Also: *Oracle Database SQL Language 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 2007, DD-rm-YYYY returns 07-ix-2007 and DD-RM-YYYY returns 07-IX-2007.

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:

Element	Description	Purpose
`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.

Loading External BFILE Data into LOB Columns

The `DBMS_LOB` PL/SQL package can load external `BFILE` data into `LOB` columns. Oracle Database performs character set conversion before loading the binary data into `CLOB` or `NCLOB` columns. Thus, the `BFILE` data does not need to be in the same character set as the database or national character set to work properly. The APIs convert the data from the specified `BFILE` character set into the database character set for the `CLOB` data type, or the national character set for the `NCLOB` data type. The loading takes place on the server because `BFILE` data is not supported on the client.

- Use `DBMS_LOB.LOADBLOBFROMFILE` to load `BLOB` columns.
- Use `DBMS_LOB.LOADCLOBFROMFILE` to load `CLOB` and `NCLOB` columns.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database SecureFiles and Large Objects Developer's Guide*

10

OCI Programming in a Global Environment

This chapter contains information about OCI programming in a globalized environment. This chapter includes the following topics:

Using the OCI NLS Functions

Many OCI NLS functions accept one of the following handles:

- The environment handle
- 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 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.

> **See Also:** *Oracle Call Interface Programmer's Guide* for detailed information about the OCI NLS functions

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 enables users to set character set information dynamically in applications, independent of the NLS_LANG and NLS_NCHAR 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 to specify UTF-16 data.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about the OCIEnvNlsCreate function

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 complies with 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 use the OCINlsGetInfo() function to retrieve the following locale information:

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

Table 10–1 summarizes OCI functions that return locale information.

Table 10–1 OCI Functions That Return Locale Information

Function	Description
OCINlsGetInfo()	Returns locale information. See preceding text.
OCINlsCharSetNameTold()	Returns the Oracle character set ID for the specified Oracle character set name
OCINlsCharSetIdToName()	Returns the Oracle character set name from the specified character set ID
OCINlsNumericInfoGet()	Returns specified numeric information such as maximum character size
OCINlsEnvironmentVariableGet( )	Returns the character set ID from NLS_LANG or the national character set ID from NLS_NCHAR

See Also: *Oracle Call Interface Programmer's Guide*

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.

Manipulating Strings in OCI

Two types of data structures are supported for string manipulation:

- Native character strings
- Wide character strings

Native character strings are encoded in native Oracle character sets. Functions that operate on native character strings take the string as a whole unit with the length of the string calculated in bytes. Wide character (`wchar`) string functions provide more flexibility in string manipulation. They support character-based and string-based operations with the length of the string calculated in characters.

The wide character data type is Oracle-specific and should not be confused with the `wchar_t` data type defined by the ANSI/ISO C standard. The Oracle wide character data type is always 4 bytes in all platforms, while the size of `wchar_t` depends on the implementation and the platform. The Oracle wide character data type normalizes native 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 format and the native character format.

String manipulation includes the:

- Conversion of strings between native character format and wide character format
- Character classifications
- Case conversion
- Calculations of display length
- General string manipulation, such as comparison, concatenation, and searching

Table 10–2 summarizes the OCI string manipulation functions.

Note: The functions and descriptions in Table 10–2 that refer to multibyte strings apply to native character strings.

Table 10–2 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.

Table 10–2 (Cont.) OCI String Manipulation Functions

Function	Description
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. **Note:** The UNICODE_BINARY sort method cannot be used with OCIWideCharStrcmp() to perform a linguistic comparison of the supplied wide character arguments.
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. **Note:** As with OCIWideCharStrcmp(), the UNICODE_BINARY sort method cannot be used with OOCIWideCharStrncmp() to perform a linguistic comparison of the supplied wide character arguments.
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.
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

See Also: *Oracle Call Interface Programmer's Guide*

Classifying Characters in OCI

Table 10–3 shows the OCI character classification functions.

Table 10–3 OCI Character Classification Functions

Function	Description
`OCIWideCharIsAlnum()`	Tests whether the wide character is an alphabetic 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 digit
`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

See Also: *Oracle Call Interface Programmer's Guide*

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 10–4 summarizes the OCI character set conversion functions.

Table 10–4 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 `OCIUnicodeToCharSet()`

See Also:

- *Oracle Call Interface Programmer's Guide*
- "OCI Programming with Unicode" on page 7-10

OCI Messaging Functions

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

Table 10–5 summarizes the OCI messaging functions.

Table 10–5 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 specified by `msgbuf`.
`OCIMessageClose()`	Closes a message handle pointed to by `msgh` and frees any memory associated with this handle

See Also: *Oracle Call Interface Programmer's Guide*

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.

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 file name convention:

```
<product_id><language_abbrev>.msb
```

For example, the file containing the server messages in French is called `oraf.msb`, because `ORA` is the product ID (`<product_id>`) and `F` is the language abbreviation (`<language_abbrev>`) for French. The value for `product_name` is rdbms, so it is in the `$ORACLE_HOME/rdbms/mesg` directory.

Syntax

```
LMSGEN text_file product facility [language] [-i indir] [-o outdir]
```

`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.
`indir` is the optional directory to specify the text file location.
`outdir` is the optional directory to specify the output file location.

The output (`.msb`) file will be generated under the `$ORACLE_HOME/product/mesg/` directory.

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 three fields:

  message_number, *warning_level*, *message_text*

  The message number must be unique within a message file.
  The warning level is not currently used. Use `0`.
  The message text cannot be longer than 511 bytes.

The following example shows an Oracle message text file:

```
/ Copyright (c) 2006 by Oracle.  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	myapp
facility	imp
language	AMERICAN
text_file	impus.msg

One of the lines in the text message file is the following:

```
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`. The directory `$ORACLE_HOME/myapp/mesg` must already exist.

```
% lmsgen impus.msg myapp imp AMERICAN
```

The following output results:

```
Generating message file impus.msg -->
$ORACLE_HOME/myapp/mesg/impus.msb

NLS Binary Message File Generation Utility: Version 10.2.0.1.0 - Production

Copyright (c) Oracle 1979, 2006.  All rights reserved.

CORE 10.2.0.1.0       Production
```

11

Character Set Migration

This chapter discusses character set conversion and character set migration. This chapter includes the following topics:

- Overview of Character Set Migration
- Changing the Database Character Set of an Existing Database
- Repairing Database Character Set Metadata
- The Language and Character Set File Scanner

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

Oracle recommends choosing Unicode for its universality and compatibility with contemporary and future technologies and language requirements. The character set defined in the Unicode Standard supports all contemporary written languages with significant use and a few historical scripts. It also supports various symbols, for example, those used in technical, scientific, and musical notations. It is the native or recommended character set of many technologies, such as Java, Windows, HTML, or XML. There is no other character set that is so universal. In addition, Unicode adoption is increasing rapidly with great support from within the industry.

Oracle's implementation of Unicode, AL32UTF8, offers encoding of ASCII characters in 1 byte, characters from European, and Middle East languages in 2 bytes, characters from South and East Asian languages in 3 bytes. Therefore, storage requirements of Unicode are usually higher than storage requirements of a legacy character set for the same language.

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**. In this case, too, Oracle recommends migrating to Unicode for its universality and compatibility. When you migrate from one database character set to another, you should also plan to minimize data loss from the following sources:

- 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 data types are specified in bytes, not characters. For example, the specification CHAR(20) in a table definition allows 20 bytes for storing character data. When the database character set uses a single-byte character encoding scheme, no data loss occurs when characters are stored because the number of characters is equivalent to the number of bytes. If the database character set uses a multibyte character set, 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 may need to be extended to support an encoding that requires multibyte storage. Truncation of data can occur if conversion causes expansion of data.

Table 11–1 shows an example of data expansion when single-byte characters become multibyte characters through conversion.

Table 11–1 Single-Byte and Multibyte Encoding

Character	WE8MSWIN 1252 Encoding	AL32UTF8 Encoding
ä	E4	C3 A4
ö	F6	C3 B6
©	A9	C2 A9
€	80	E2 82 AC

The first column of Table 11–1 shows selected characters. The second column shows the hexadecimal representation of the characters in the WE8MSWIN1252 character set. The third column shows the hexadecimal representation of each character in the AL32UTF8 character set. Each pair of letters and numbers represents one byte. For example, ä (a with an umlaut) is a single-byte character (E4) in WE8MSWIN1252, but it becomes a two-byte character (C3 A4) in AL32UTF8. Also, the encoding for the euro symbol expands from one byte (80) to three bytes (E2 82 AC).

If the data in the new character set requires storage that is greater than the supported byte size of the data types, then you must change your schema. You may need to use CLOB columns.

See Also: "Length Semantics" on page 2-9

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. 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 AL32UTF8, 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 AL32UTF8.
- If existing Oracle usernames or passwords are created based on characters that change in size in the new character set, then users will have trouble logging in

because of authentication failures after the migration to a new character set. This occurs because the encrypted usernames and passwords stored in the data dictionary may not be 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 AL32UTF8, then the length of the username `scött` (`o` with an umlaut) changes from 5 bytes to 6 bytes. In AL32UTF8, `scött` can no longer log in because of the difference in the username. Oracle recommends that usernames and passwords be based on ASCII characters. If they are not, then you must reset the affected usernames and passwords after migrating to a new character set.

- When `CHAR` data contains characters that expand after migration to a new character set, space padding is not 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: *Oracle 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
- Conversion from Single-byte to Multibyte Character Set and Oracle Data Pump

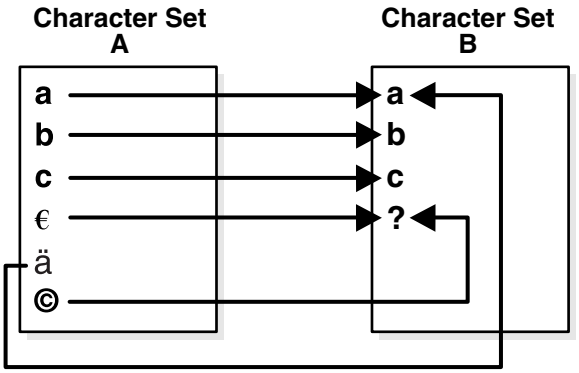
Replacement Characters that Result from Using the Export and Import Utilities

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, then the destination character set B should be a superset of character set A. The destination character set, 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 as 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.

Note: There is an exception to the requirement that the destination character set B should be a superset of character set A. If your data contains no characters that are in character set A but are not in character set B, then the destination character set does not need to be a superset of character set A to avoid data loss or data corruption.

Figure 11–1 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 11–1 Replacement Characters in Character Set Conversion



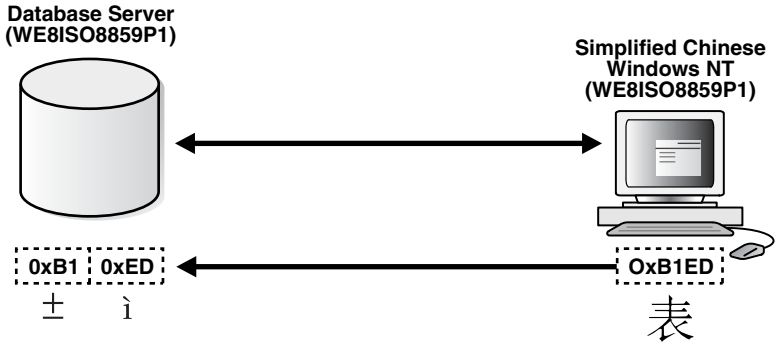
To reduce the risk of losing data, choose a destination character set with a similar character repertoire. Migrating to Unicode may be the best option, because AL32UTF8 contains characters from most legacy character sets.

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 AL32UTF8, the client is an English Windows operating system, and the `NLS_LANG` setting on the client is AL32UTF8. Data coming into the database is encoded in WE8MSWIN1252 and is not converted to AL32UTF8 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.

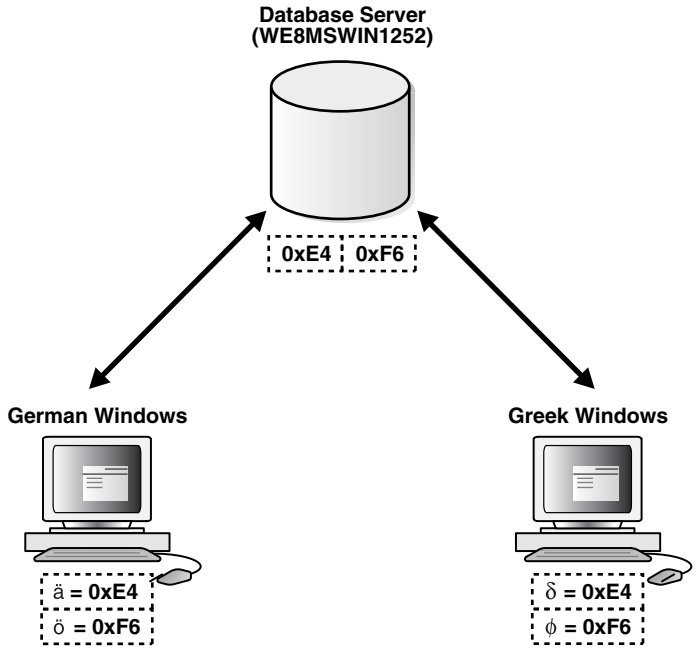
This can lead to two possible data inconsistency problems. One problem occurs when a database contains data from a character set that is different from the database character set but the same code points exist in both character sets. For example, if the database character set is WE8ISO8859P1 and the `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 treats these characters as single-byte WE8ISO8859P1 characters. Hence all SQL string manipulation functions such as SUBSTR or LENGTH are 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 such as AL32UTF8, then character codes are converted as if they were in WE8ISO8859P1. This way, each of the two bytes of a ZHS16GBK character are converted separately, yielding meaningless values in AL32UTF8. Figure 11–2 shows an example of this incorrect character set replacement.

Figure 11–2 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 WE8MSWIN1252 as the `NLS_LANG` character set, then the database contains a mixture of German and Greek characters. Figure 11–3 shows how different clients can use different character sets in the same database.

Figure 11–3 Mixed Character Sets



For database character set migration to be successful, both of these cases require manual intervention because Oracle Database cannot determine the character sets of the data being stored. 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.

Conversion from Single-byte to Multibyte Character Set and Oracle Data Pump

If Oracle Data Pump is being used, and if a character set migration from single-byte to multibyte is performed, then the Data Pump PL/SQL packages must be reloaded.

Changing the Database Character Set of an Existing Database

Database character set migration is an intricate process that typically involves three stages: data scanning, data cleansing, and data conversion.

Before you change the database character set, you must 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.

After the potential data issues are identified, they need to be cleansed properly to ensure the data integrity can be preserved during the data conversion. The data cleansing step could require significant time and effort depending on the scale and complexity of the data issues found. It may take multiple iterations of data scanning and cleansing in order to correctly address all of the data exceptions.

The data conversion is the process by which the character data is converted from the source character set into the target character set representation. 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.

There are two approaches for migrating the database character set:

- Migrating Character Data Using the Database Migration Assistant for Unicode
- Migrating Character Data Using a Full Export and Import

Migrating Character Data Using the Database Migration Assistant for Unicode

The Database Migration Assistant for Unicode (DMU) offers an intuitive and user-friendly GUI that helps you streamline the migration process to Unicode through an interface that minimizes the manual workload and ensures that the migration tasks are carried out correctly and efficiently.

Some advantages of the DMU are that it does the following:

- Guides you through the workflow

  An important advantage of the DMU is that it offers a logical workflow to guide you through the entire process of migrating character sets.

- Offers suggestions for handling certain problems

  The DMU can help you when you run into certain problems, such as errors or failures during the scanning or cleansing of the data.

- Supports selective conversion of data

  The DMU enables you to convert only the data that must be converted, at the table, column, and row level.

- Offers progress monitoring

  The DMU provides a GUI to visualize how the steps are progressing.

- Offers interactive visualization features

  The DMU enables you to analyze data and see the results in the GUI in an interactive way. It also enables you to see the data itself in the GUI and cleanse it interactively from identified migration issues.

- Provides the only supported tool for inline conversion

  With the DMU, Oracle Database supports inline conversion of database contents. This offers performance and security advantage over other existing conversion methods.

- Allows cleansing actions to be scheduled for later execution during the conversion step

  Postponing of cleansing actions, such as data type migration, ensures that the production database and applications are not affected until the actual migration downtime window.

This release of the Database Migration Assistant for Unicode has a few restrictions with respect to what databases it can convert. In particular, it does not convert databases with certain types of convertible data in the data dictionary. The export/import migration methods could be used to overcome these limitations.

In the current database release, the DMU is installed under the `$ORACLE_HOME/dmu` directory.

See Also: *Oracle Database Migration Assistant for Unicode Guide*

Migrating Character Data Using a Full Export and Import

A full export and import can also be used to convert the database to a new character set. It may be more time-consuming and resource-intensive as a separate target instance must be set up. If you plan to migrate your data to a non-Unicode character set, which Oracle strongly discourages, you can use the DMU to look for invalid character representation issues in the database and use export and import for the data conversion. Note that the DMU will not correctly identify data expansion issues (column and data type limit violations) if the migration is not to Unicode. It will also not identify characters that exist in the source database character set but do not exist in the non-Unicode target character set.

See Also: *Oracle Database Utilities* for more information about the Export and Import utilities

Repairing Database Character Set Metadata

If your database has been in what is commonly called a pass-through configuration, where the client character set is defined (usually through the `NLS_LANG` client setting) to be equal to the database character set, the character data in your database could be stored in a different character set from the declared database character set. In this scenario, the recommended solution is to migrate your database to Unicode by using the DMU assumed database character set feature to indicate the actual character set for the data. In case migrating to Unicode is not immediately feasible due to business or technical constraints, it would be desirable to at least correct the database character set declaration to match with the database contents.

With Database Migration Assistant for Unicode Release 1.2, you can repair the database character set metadata in such cases using the `CSREPAIR` script. The `CSREPAIR` script works in conjunction with the DMU client and accesses the DMU repository. It can be used to change the database character set declaration to the real character set of the data only after the DMU has performed a full database scan by setting the Assumed Database Character Set property to the target character set and no invalid representation issues have been reported, which verifies that all existing data in the database is defined according to the assumed database character set. Note that

CSREPAIR only changes the database character set declaration in the data dictionary metadata and does not convert any database data.

You can find the CSREPAIR script under the admin subdirectory of the DMU installation. The requirements when using the CSREPAIR script are:

1. You must first perform a successful full database scan in the DMU with the Assumed Database Character Set property set to the real character set of the data. In this case, the assumed database character set must be different from the current database character set or else nothing will be done. The CSREPAIR script will not proceed if the DMU reports the existence of invalid data. It will, however, proceed if changeless or convertible data is present from the scan.
2. The target character set in the assumed database character set must be a binary superset of US7ASCII.
3. Only repairing from single-byte to single-byte character sets or multi-byte to multi-byte character sets is allowed as no conversion of CLOB data will be attempted.
4. If you set the assumed character set at the column level, then the value must be the same as the assumed database character set. Otherwise, CSREPAIR will not run.
5. You must have the SYSDBA privilege to run CSREPAIR.

Example: Using CSREPAIR

A typical example is storing WE8MSWIN1252 data in a WE8ISO8859P1 database via the pass-through configuration. To correct the database character set from WE8ISO8859P1 to WE8MSWIN1252, perform the following steps:

1. Set up the DMU and connect to the target WE8ISO8859P1 database.
2. Open the Database Properties tab in the DMU.
3. Set the Assumed Database Character Set property to WE8MSWIN1252.
4. Use the DMU to perform a full database scan.
5. Open the Database Scan Report and verify there is no data reported under the Invalid Representation category.
6. Exit from the DMU client.
7. Start the SQL*Plus utility and connect as a user with the SYSDBA privilege.
8. Run the CSREPAIR script:

   ```
   SQL> @@CSREPAIR.PLB
   ```

   Upon completion, you should get the message:

   The database character set has been successfully changed to WE8MSWIN1252. You must restart the database now.
9. Shut down and restart the database.

The Language and Character Set File Scanner

The Language and Character Set File Scanner (LCSSCAN) is a high-performance, statistically based utility for determining the language and character set for unknown file text. It can automatically identify a wide variety of language and character set pairs. With each text, the language and character set detection engine sets up a series

of probabilities, each probability corresponding to a language and character set pair. The most statistically probable pair identifies the dominant language and character set.

The purity of the text affects the accuracy of the language and character set detection. The ideal case is literary text of one single language with no spelling or grammatical errors. These types of text may require 100 characters of data or more and can return results with a very high factor of confidence. On the other hand, some technical documents can require longer segments before they are recognized. Documents that contain a mix of languages or character sets or text such as addresses, phone numbers, or programming language code may yield poor results. For example, if a document has both French and German embedded, then the accuracy of guessing either language successfully is statistically reduced. Both plain text and HTML files are accepted. If the format is known, you should set the `FORMAT` parameter to improve accuracy.

This section includes the following topics:

Syntax of the LCSSCAN Command

Start the Language and Character Set File Scanner with the `LCSSCAN` command. Its syntax is as follows:

```
LCSSCAN  [RESULTS=number] [FORMAT=file_type] [BEGIN=number] [END=number]
FILE=file_name
```

The parameters are described in the rest of this section.

RESULTS

The `RESULTS` parameter is optional.

Property	Description
Default value	`1`
Minimum value	`1`
Maximum value	`3`
Purpose	The number of language and character set pairs that are returned. They are listed in order of probability. The comparative weight of the first choice cannot be quantified. The recommended value for this parameter is the default value of `1`.

FORMAT

The `FORMAT` parameter is optional.

Property	Description
Default Value	`text`
Purpose	This parameter identifies the type of file to be scanned. The possible values are `html`, `text`, and `auto`.

BEGIN

The BEGIN parameter is optional.

Property	Description
Default value	1
Minimum value	1
Maximum value	Number of bytes in file
Purpose	The byte of the input file where LCSSCAN begins the scanning process. The default value is the first byte of the input file.

END

The END parameter is optional.

Property	Description
Default value	End of file
Minimum value	3
Maximum value	Number of bytes in file
Purpose	The last byte of the input file that LCSSCAN scans. The default value is the last byte of the input file.

FILE

The FILE parameter is required.

Property	Description
Default value	None
Purpose	Specifies the name of a text file to be scanned

Examples: Using the LCSSCAN Command

Example 11–1 Specifying Only the File Name in the LCSSCAN Command

```
LCSSCAN FILE=example.txt
```

In this example, the entire example.txt file is scanned because the BEGIN and END parameters have not been specified. One language and character set pair will be returned because the RESULTS parameter has not been specified.

Example 11–2 Specifying the Format as HTML

```
LCSSCAN FILE=example.html FORMAT=html
```

In this example, the entire example.html file is scanned because the BEGIN and END parameters have not been specified. The scan will strip HTML tags before the scan, thus results are more accurate. One language and character set pair will be returned because the RESULTS parameter has not been specified.

Example 11–3 Specifying the RESULTS and BEGIN Parameters for LCSSCAN

```
LCSSCAN RESULTS=2 BEGIN=50 FILE=example.txt
```

The scanning process starts at the 50th byte of the file and continues to the end of the file. Two language and character set pairs will be returned.

Example 11–4 Specifying the RESULTS and END Parameters for LCSSCAN

```
LCSSCAN RESULTS=3 END=100 FILE=example.txt
```

The scanning process starts at the beginning of the file and ends at the 100th byte of the file. Three language and character set pairs will be returned.

Example 11–5 Specifying the BEGIN and END Parameters for LCSSCAN

```
LCSSCAN BEGIN=50 END=100 FILE=example.txt
```

The scanning process starts at the 50th byte and ends at the 100th byte of the file. One language and character set pair will be returned because the `RESULTS` parameter has not been specified.

Getting Command-Line Help for the Language and Character Set File Scanner

To obtain a summary of the Language and Character Set File Scanner parameters, enter the following command:

```
LCSSCAN HELP=y
```

The resulting output shows a summary of the Language and Character Set Scanner parameters.

Supported Languages and Character Sets

The Language and Character Set File Scanner supports several character sets for each language.

When the binary values for a language match two or more encodings that have a subset/superset relationship, the subset character set is returned. For example, if the language is German and all characters are 7-bit, then US7ASCII is returned instead of WE8MSWIN1252, WE8ISO8859P15, or WE8ISO8859P1.

When the character set is determined to be UTF-8, the Oracle character set UTF8 is returned by default unless 4-byte characters (supplementary characters) are detected within the text. If 4-byte characters are detected, then the character set is reported as AL32UTF8.

> **See Also:** "Language and Character Set Detection Support" on page A-19 for a list of supported languages and character sets

LCSSCAN Error Messages

LCD-00001 An unknown error occured.

Cause: An error occurred accessing an internal structure.

Action: Report this error to Oracle Support.

LCD-00002 NLS data could not be loaded.

Cause: An error occurred accessing `$ORACLE_HOME/nls/data`.

Action: Check to make sure `$ORACLE_HOME/nls/data` exists and is accessible. If not found check `$ORA_NLS10` directory.

LCD-00003 An error occurred while reading the profile file.

Cause: An error occurred accessing `$ORACLE_HOME/nls/data`.

Action: Check to make sure `$ORACLE_HOME/nls/data` exists and is accessible. If not found check `$ORA_NLS10` directory.

LCD-00004 The beginning or ending offset has been set incorrectly.

Cause: The beginning and ending offsets must be an integer greater than 0.

Action: Change the offset to a positive number.

LCD-00005 The ending offset has been set incorrectly.

Cause: The ending offset must be greater than the beginning offset.

Action: Change the ending offset to be greater than the beginning offset.

LCD-00006 An error occurred when opening the input file.

Cause: The file was not found or could not be opened.

Action: Check the name of the file specified. Make sure the full file name is specified and that the file is not in use.

LCD-00007 The beginning offset has been set incorrectly.

Cause: The beginning offset must be less than the number of bytes in the file.

Action: Check the size of the file and specify a smaller beginning offset.

LCD-00008 No result was returned.

Cause: Not enough text was inputted to produce a result.

Action: A larger sample of text needs to be inputted to produce a reliable result.

12

Customizing Locale Data

This chapter describes how to customize locale data and 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 manages 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 uses Unicode characters in many of its functions. For example, it shows the mapping of local character code points to Unicode code points. Oracle Locale Builder depends on the local fonts that are available on the operating system where the characters are rendered. Therefore, Oracle 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 recommends using the Arial Unicode MS font from Microsoft, because it includes over 50,000 glyphs and supports most of the characters in Unicode 6.2.

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/jre/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 uses the first font available on your system.

After you edit the `font.properties` file, restart the Oracle Locale Builder.

> **See Also:** Sun's internationalization Web site 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
```

> **Note:** Depending on the operating system locale, the locale-specific `font.properties` file might be used. For example, if the current operating system locale is `ja_JP.eucJP` on Sun Solaris, then `font.properties.ja` may be used.

> **See Also:** Your operating system documentation for more information about available fonts

The Oracle Locale Builder User Interface

Ensure that the `ORACLE_HOME` parameter is set before starting Oracle Locale Builder.

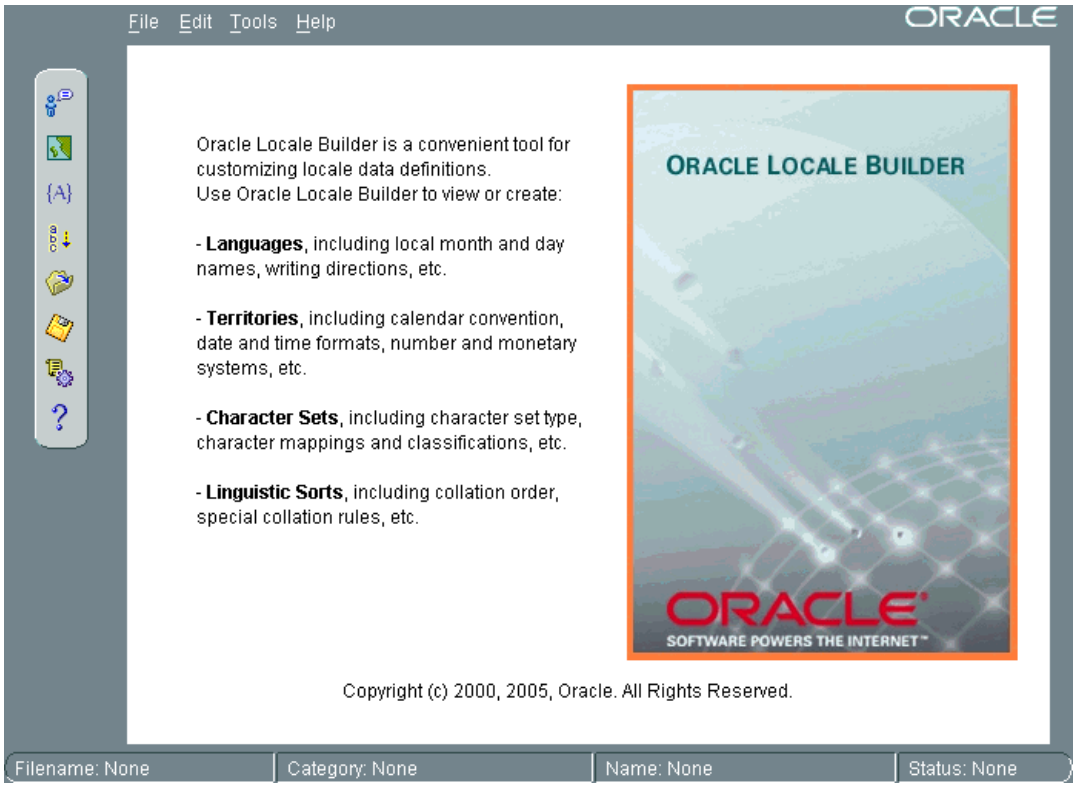
In the UNIX operating system, start the Oracle Locale Builder by changing into the `$ORACLE_HOME/nls/lbuilder` directory and issuing the following command:

```
% ./lbuilder
```

In a Windows operating system, start the Oracle Locale Builder from the Start menu as follows: **Start** > **Programs** > **Oracle-OraHome10** > **Configuration and Migration Tools** > **Locale Builder**. You can also start it from the DOS prompt by entering the `%ORACLE_HOME%\nls\lbuilder` directory and executing the `lbuilder.bat` command.

When you start the Oracle Locale Builder, the screen shown in Figure 12–1 appears.

Figure 12–1 Oracle Locale Builder Utility



Oracle Locale Builder Pages and Dialog Boxes

Before using Oracle Locale Builder for a specific task, you should become familiar with the following tab pages and dialog boxes:

- Existing Definitions Dialog Box
- Session Log Dialog Box
- Preview NLT Tab Page
- Open File Dialog Box

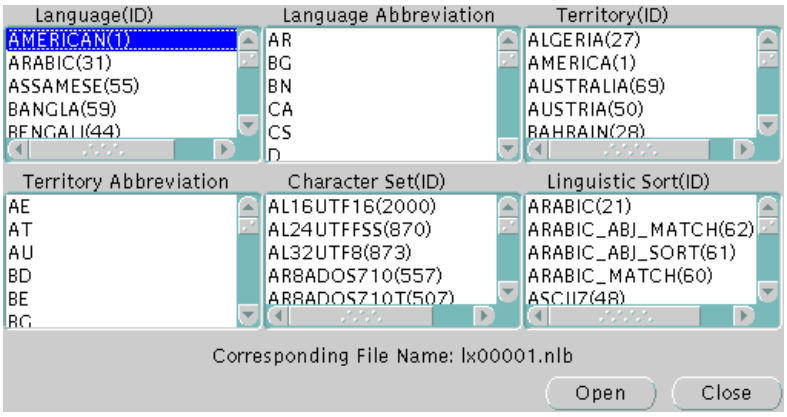
> **Note:** Oracle Locale Builder includes online help.

Existing Definitions Dialog Box

When you choose **New Language**, **New Territory**, **New Character Set**, or **New Linguistic Sort**, the first tab page that you see is labeled **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, then 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



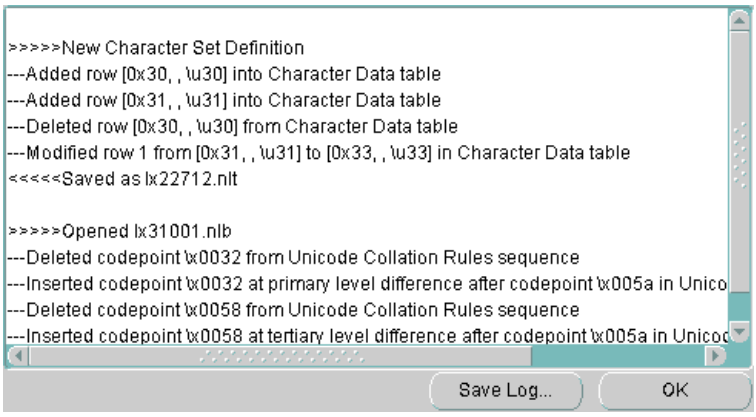
Choosing `AMERICAN` opens the `lx00001.nlb` file. An NLB file is a binary file that contains the settings for a specific language, territory, character set, or linguistic sort.

Language and territory abbreviations are for reference only and cannot be opened.

Session Log Dialog Box

Choose **Tools** > **View Log** to see the Session Log dialog box. The Session Log dialog box shows what actions have been taken in the current session. Click **Save Log** to keep a record of all changes. Figure 12–3 shows an example of a session log.

Figure 12–3 Session Log Dialog Box



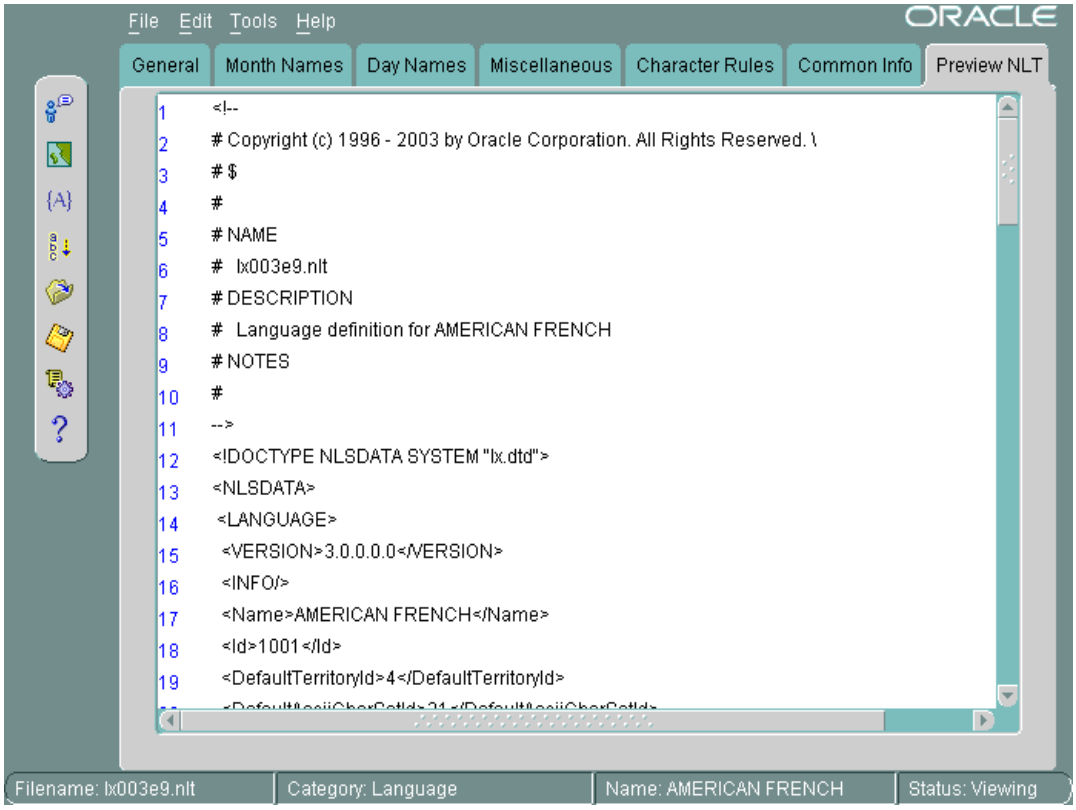
Preview NLT Tab Page

The NLT (National Language Text) file is an XML file with the file extension `.nlt` that stores the settings for a specific language, territory, character set, or linguistic sort. The

Preview NLT tab page presents a readable form of the file so that you can see whether the changes you have made are correct. You cannot modify the NLT file from the **Preview NLT** tab page. You must use the specific tools and procedures available in Oracle Locale Builder to modify the NLT file.

Figure 12–4 shows an example of the **Preview NLT** tab page for a user-defined language called `AMERICAN FRENCH`.

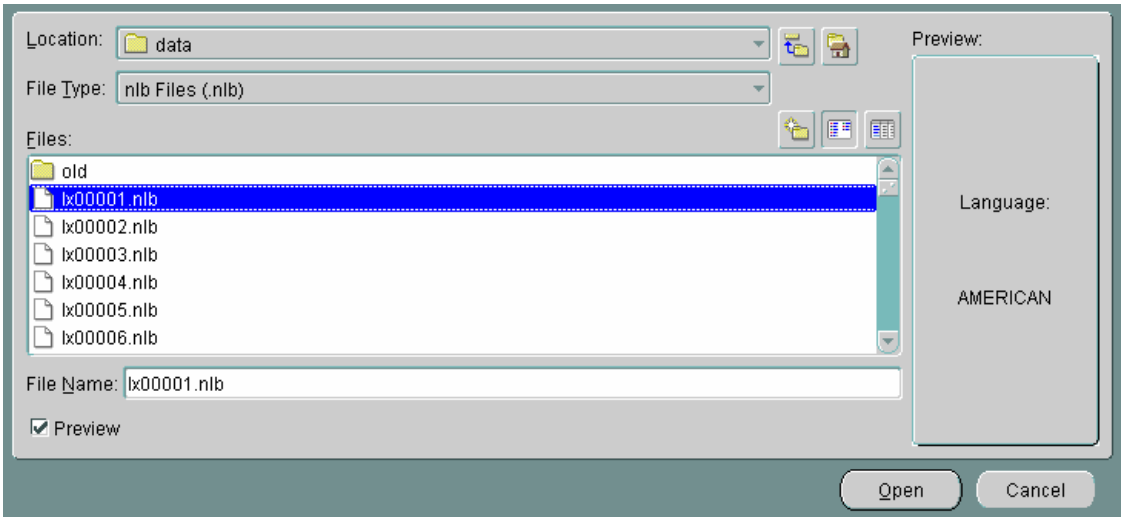
Figure 12–4 Previewing the NLT File



Open File Dialog Box

You can see the Open File dialog box by choosing **File** > **Open** > **By File Name**. Then choose the NLB (National Language Binary) 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** pane shows that this NLB file is for the `AMERICAN` language.

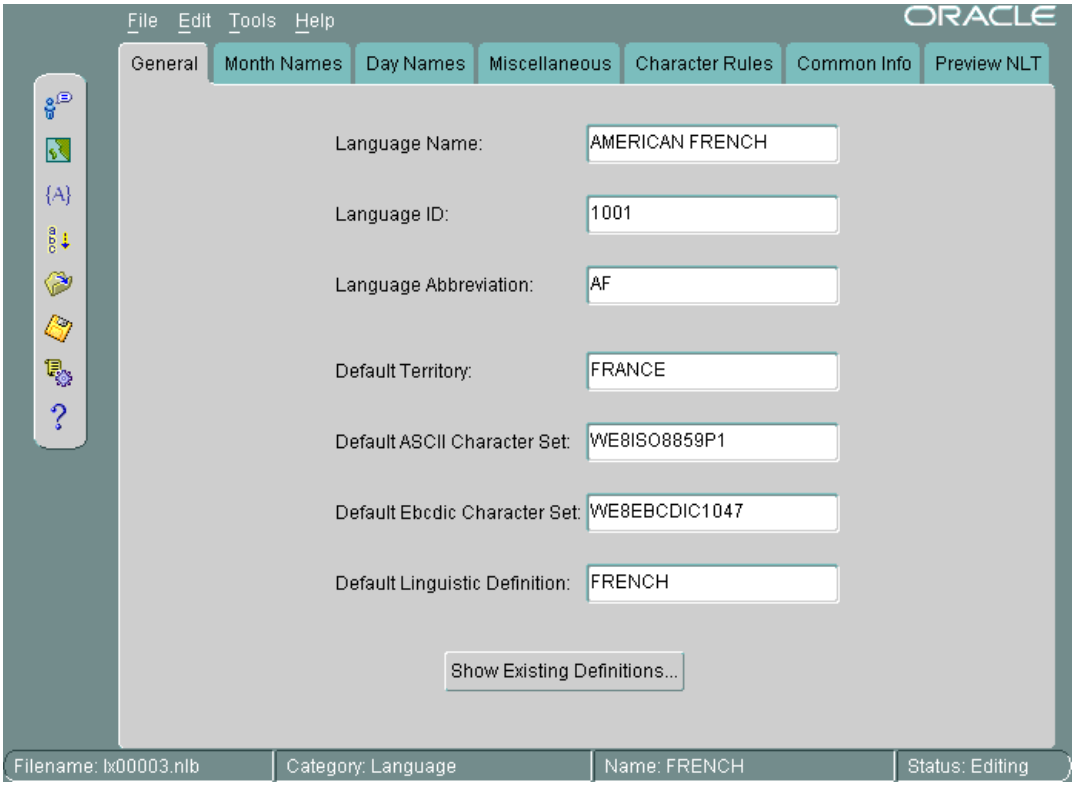
Figure 12–5 Open File Dialog Box



Creating a New Language Definition with 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** tab page. Retain the default values for the other fields. Figure 12–6 shows the resulting **General** tab page.

Figure 12–6 Language General Information



The following restrictions apply when choosing names for locale objects such as languages:

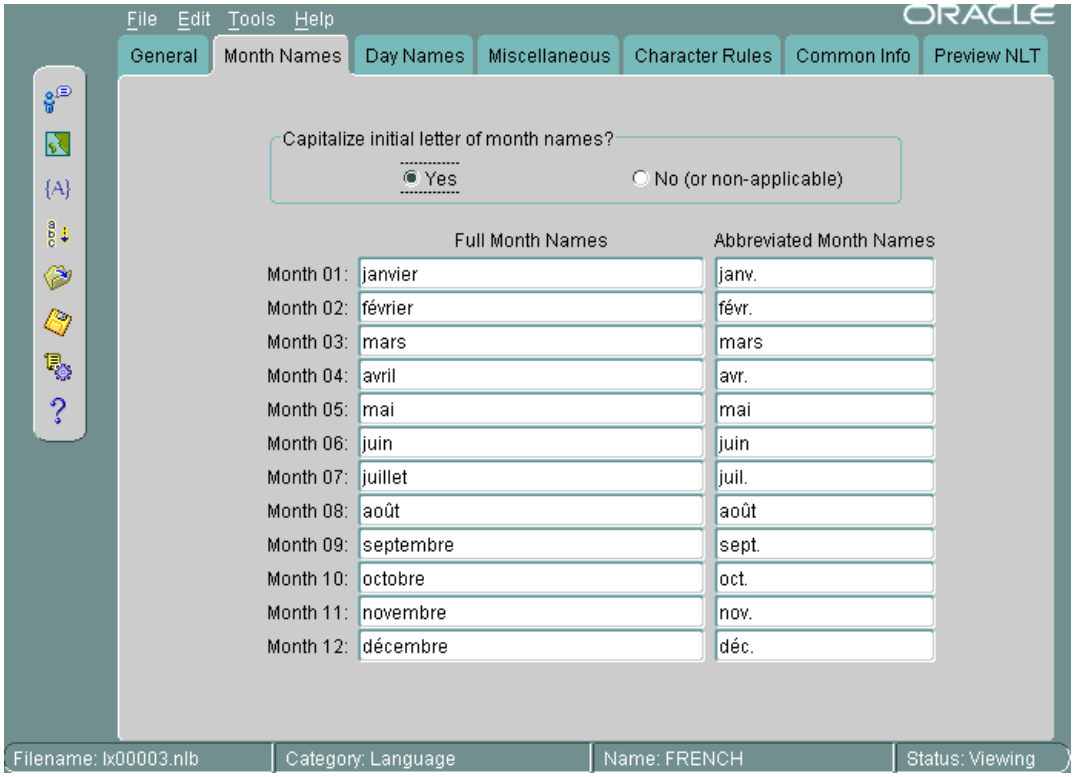
- Names must contain only ASCII characters
- Names must start with a letter and cannot have leading or trailing blanks
- Language, territory, and character set names cannot contain underscores or periods

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 page.

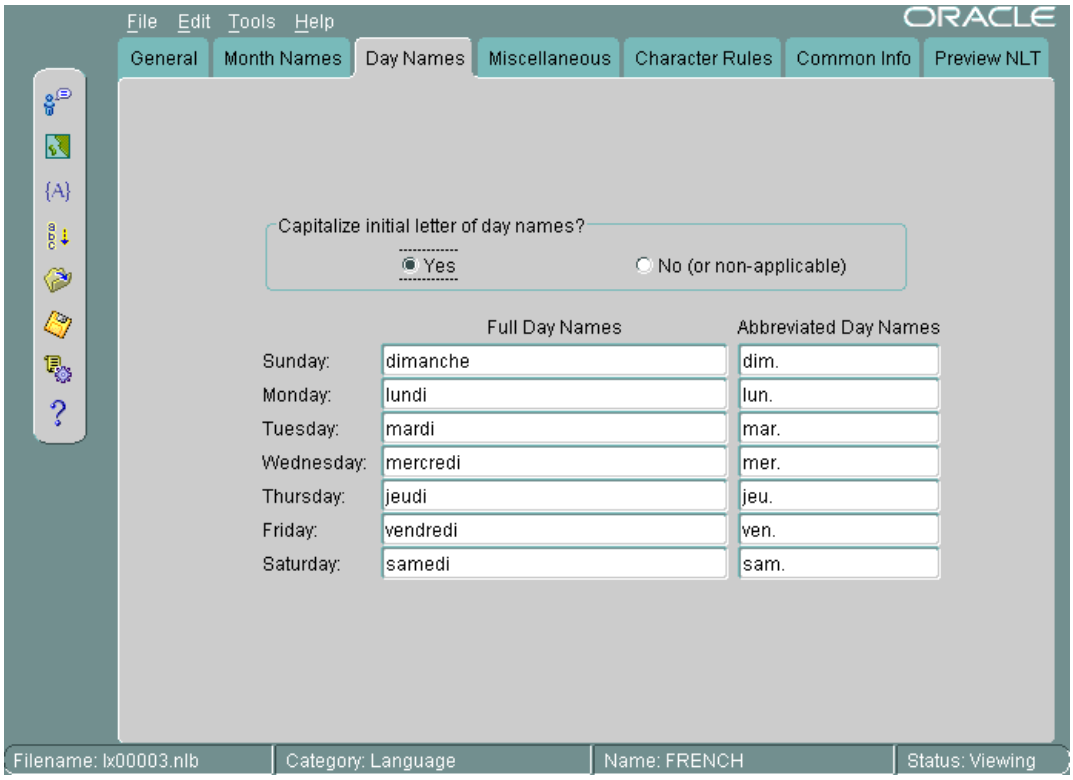
Figure 12–7 Month Names Tab Page



All names are shown as they appear in the NLT file. If you choose **Yes** for capitalization, then the month names are capitalized in your application, but they do not appear capitalized in the **Month Names** tab page.

Figure 12–8 shows the **Day Names** tab page.

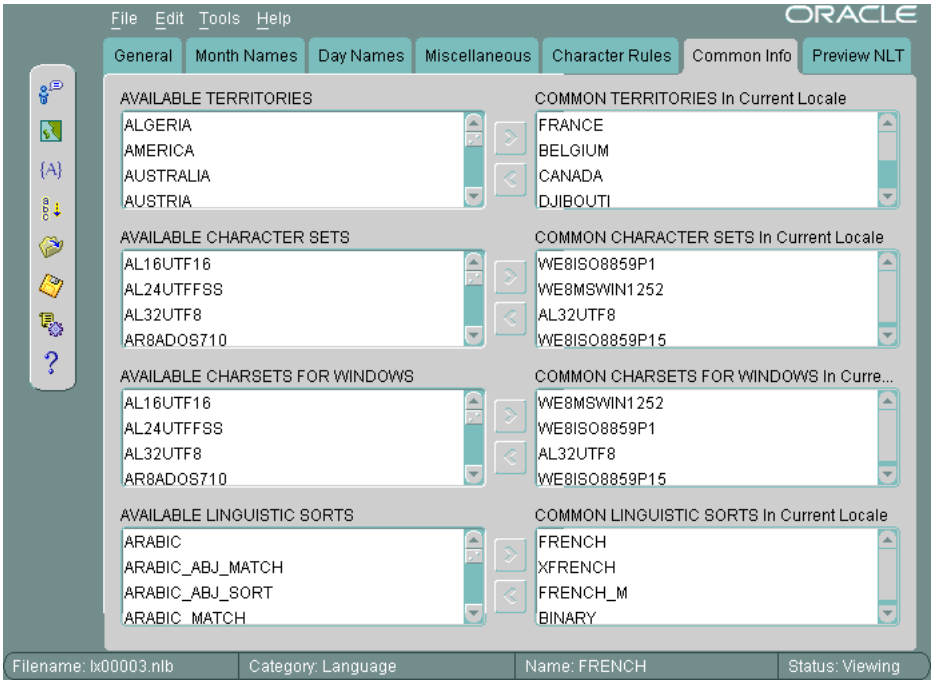
Figure 12–8 Day Names Tab Page



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, then the day names are capitalized in your application, but they do not appear capitalized in the **Day Names** tab page.

Figure 12–9 shows the **Common Info** tab page.

Figure 12–9 Common Info Tab Page



You can display the territories, character sets, Windows character sets, and linguistic sorts that have associations with the current language. In general, the most appropriate or the most commonly used items are displayed first. For example, with a language of FRENCH, the common territories are FRANCE, BELGIUM, CANADA, and DJIBOUTI, while the character sets for supporting French are WE8ISO8859P1, WE8MSWIN1252, AL32UTF8, and WE8ISO8859P15. As WE8MSWIN1252 is more common than WE8ISO8859P1 in a Windows environment, it is displayed first.

Creating a New Territory Definition with the Oracle Locale Builder

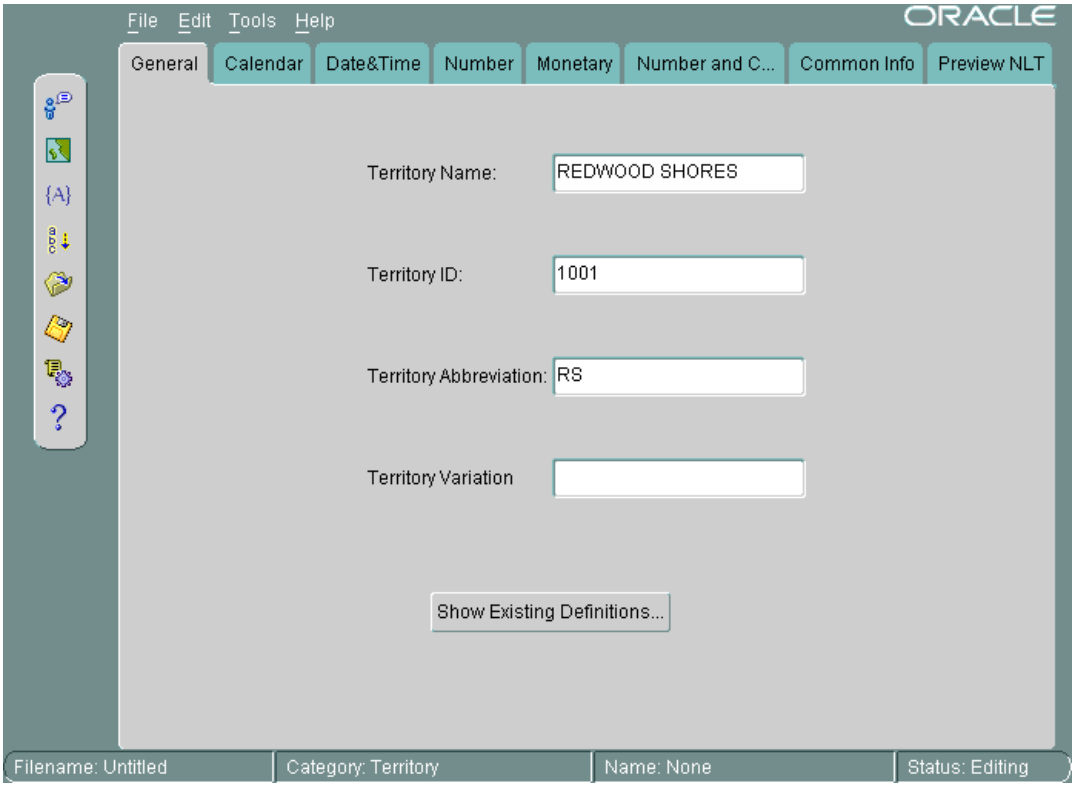
This section shows how to create a new territory called `REDWOOD SHORES` and use `RS` as a territory abbreviation. The new territory is not based on an existing territory definition.

The basic tasks are as follows:

- Assign a territory name
- Choose formats for the calendar, numbers, date and time, and currency

Figure 12–10 shows the **General** tab page with `REDWOOD SHORES` specified as the **Territory Name**, `1001` specified as the **Territory ID**, and `RS` specified as the **Territory Abbreviation**.

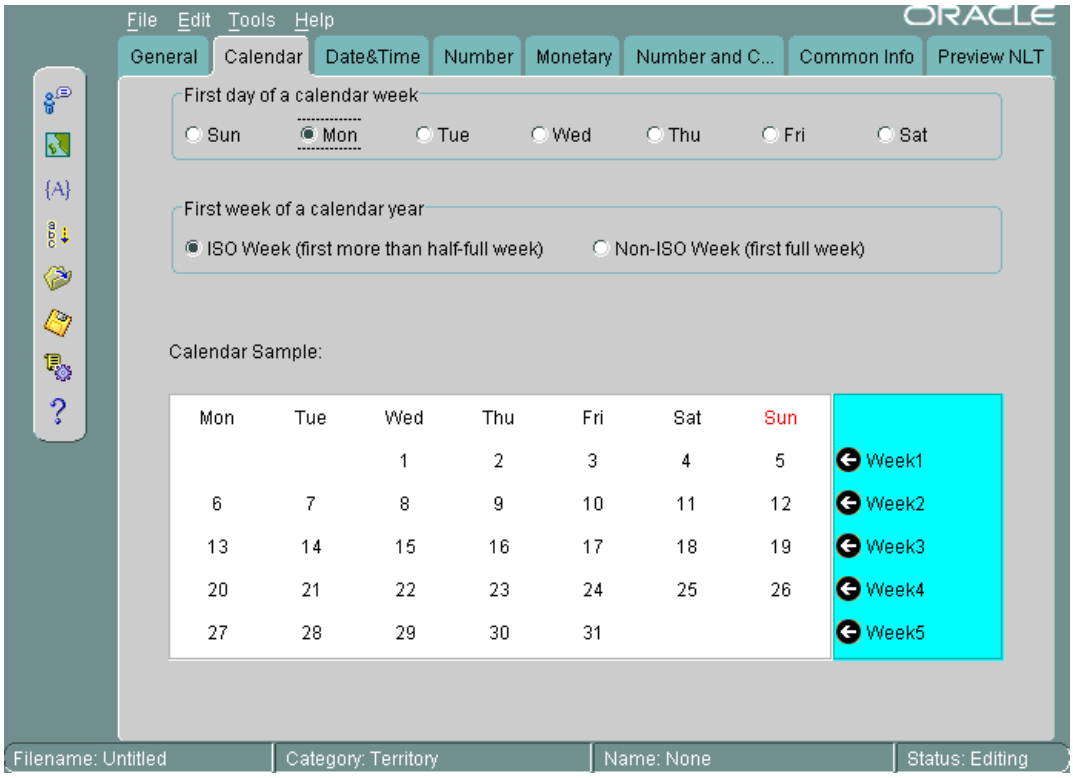
Figure 12–10 General Tab Page for Territories



The valid range for **Territory ID** for a user-defined territory is 1000 to 10000.

Figure 12–11 shows settings for calendar formats in the **Calendar** tab page.

Figure 12–11 Choosing Calendar Formats



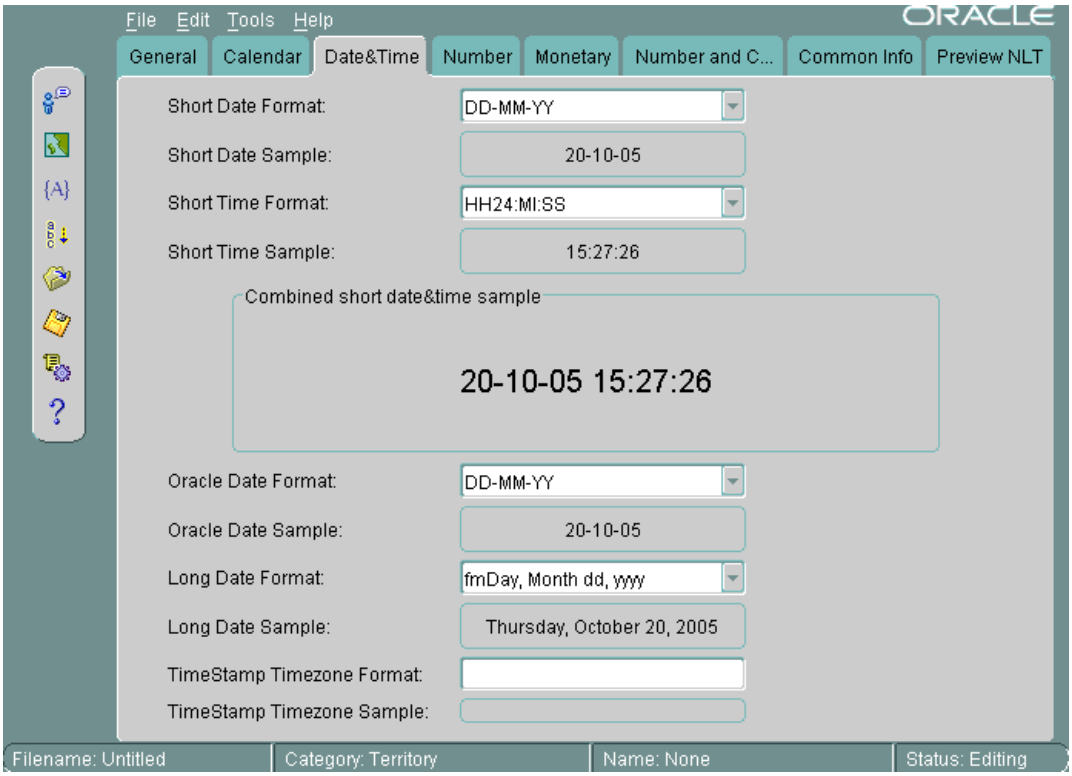
Monday is set as the first day of the week, and the first week of the calendar year is set as an ISO week. Figure 12–11 displays a sample calendar.

See Also:

- "Calendar Formats" on page 3-20 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 12-36 for information about customizing calendars themselves

Figure 12–12 shows the **Date&Time** tab page.

Figure 12–12 Choosing Date and Time Formats



When you choose a format from a list, Oracle Locale Builder displays an example of the format. In this case, the **Short Date Format** is set to `DD-MM-YY`. The **Short Time Format** is set to `HH24:MI:SS`. The **Oracle Date Format** is set to `DD-MM-YY`. The **Long Date Format** is set to `fmDay, Month dd, yyyy`. The **TimeStamp Timezone Format** is not set.

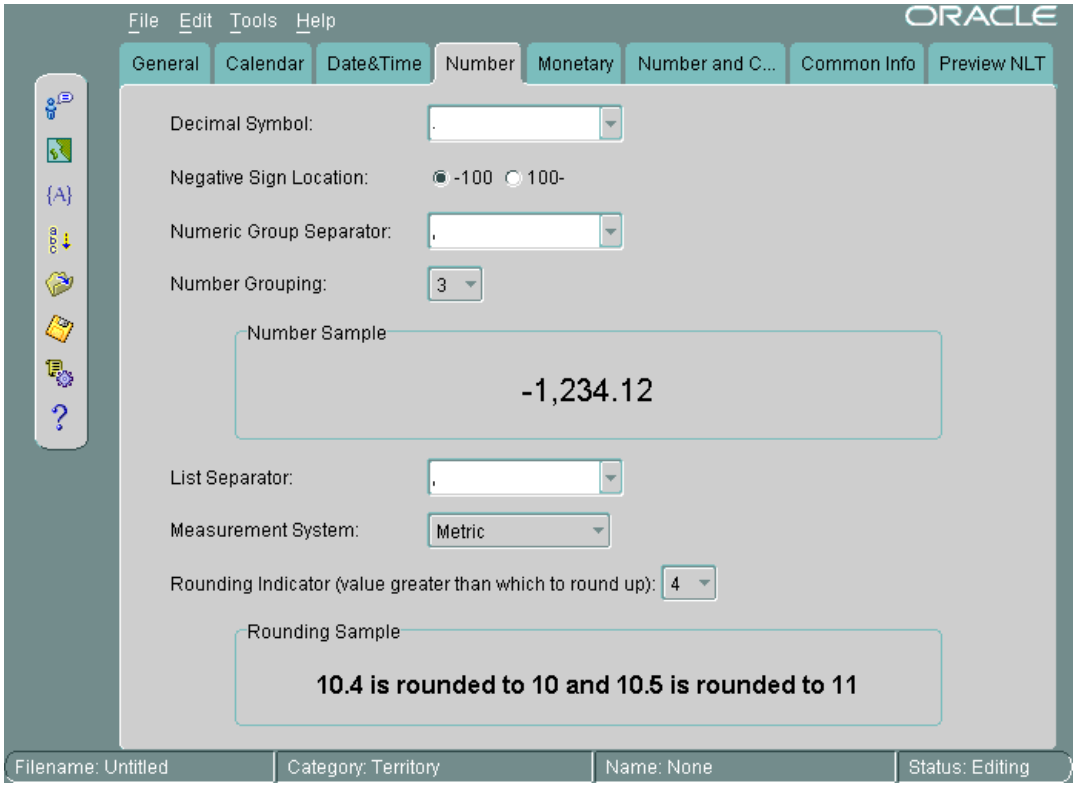
You can also enter your own formats instead of using the selection from the drop-down menus.

See Also:

- "Date Formats" on page 3-15
- "Time Formats" on page 3-18
- "Choosing a Time Zone File" on page 4-15

Figure 12–13 shows the **Number** tab page.

Figure 12–13 Choosing Number Formats



A period has been chosen for the **Decimal Symbol**. The **Negative Sign Location** is specified to be on the left of the number. The **Numeric Group Separator** is a comma. The **Number Grouping** is specified as 3 digits. The **List Separator** is a comma. The **Measurement System** is metric. The **Rounding Indicator** is 4.

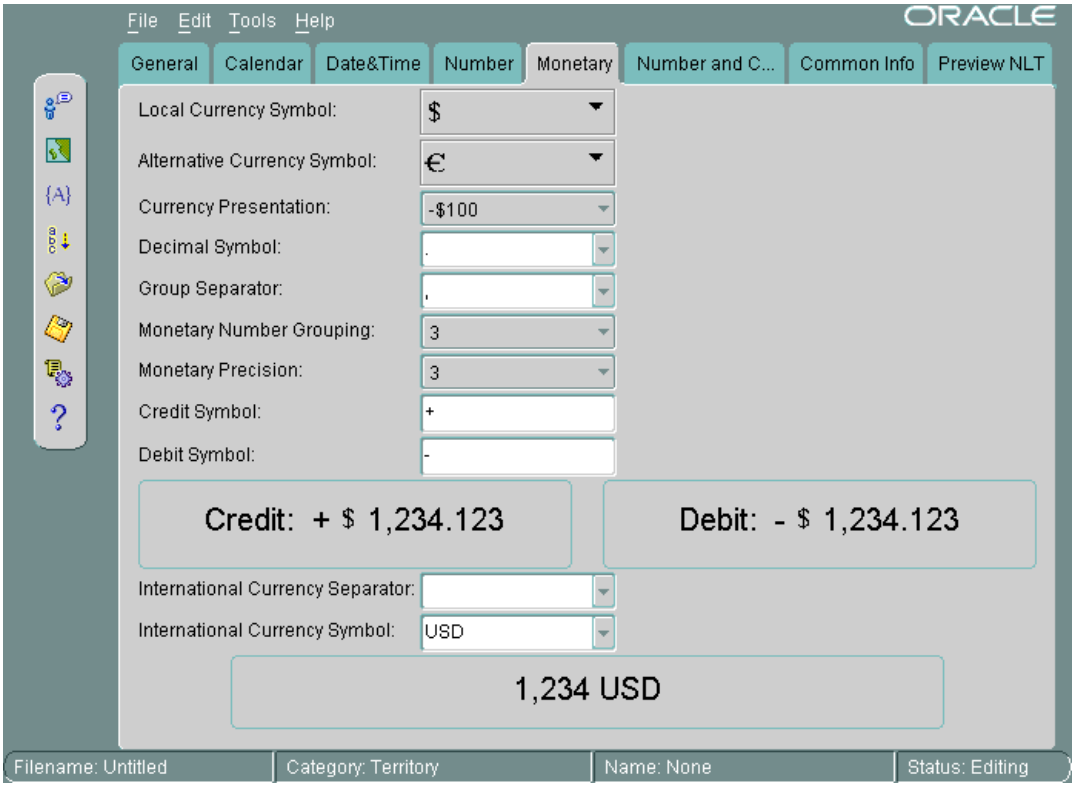
You can enter your own values instead of using values in the lists.

When you choose a format from a list, Oracle Locale Builder displays an example of the format.

> **See Also:** "Numeric Formats" on page 3-23

Figure 12–14 shows settings for currency formats in the **Monetary** tab page.

Figure 12–14 Choosing Currency Formats



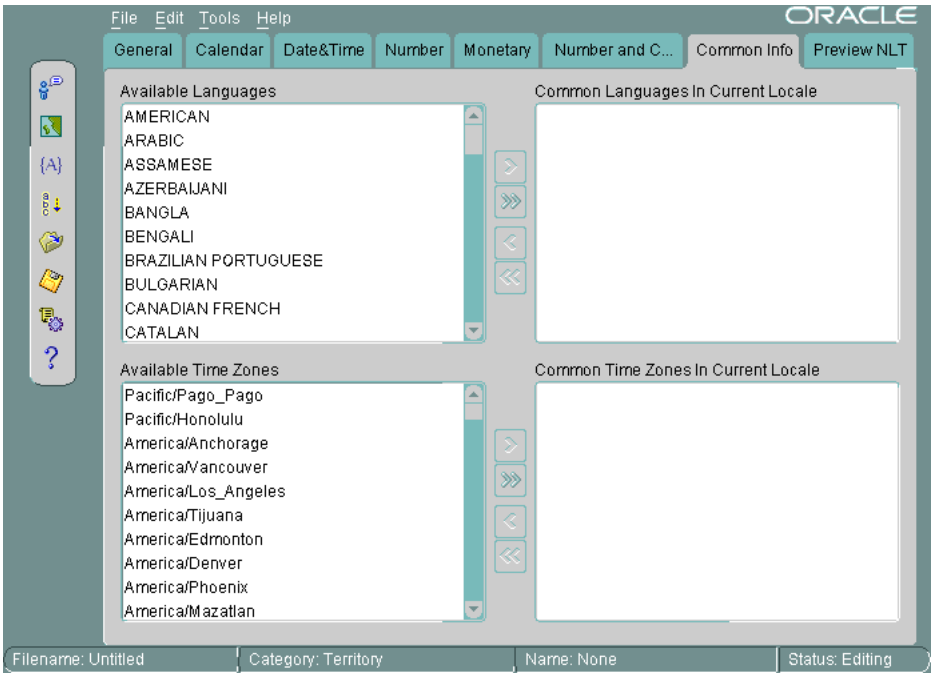
The **Local Currency Symbol** is set to `$`. The **Alternative Currency Symbol** is the euro symbol. The **Currency Presentation** shows one of several possible sequences of the local currency symbol, the debit symbol, and the number. The **Decimal Symbol** is the period. The **Group Separator** is the 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 field. The **International Currency Symbol** (ISO currency symbol) is `USD`. Oracle Locale Builder displays examples of the currency formats you have selected.

You can enter your own values instead of using the lists.

See Also: "Currency Formats" on page 3-25

Figure 12–15 shows the **Common Info** tab page.

Figure 12–15 Common Info Tab Page

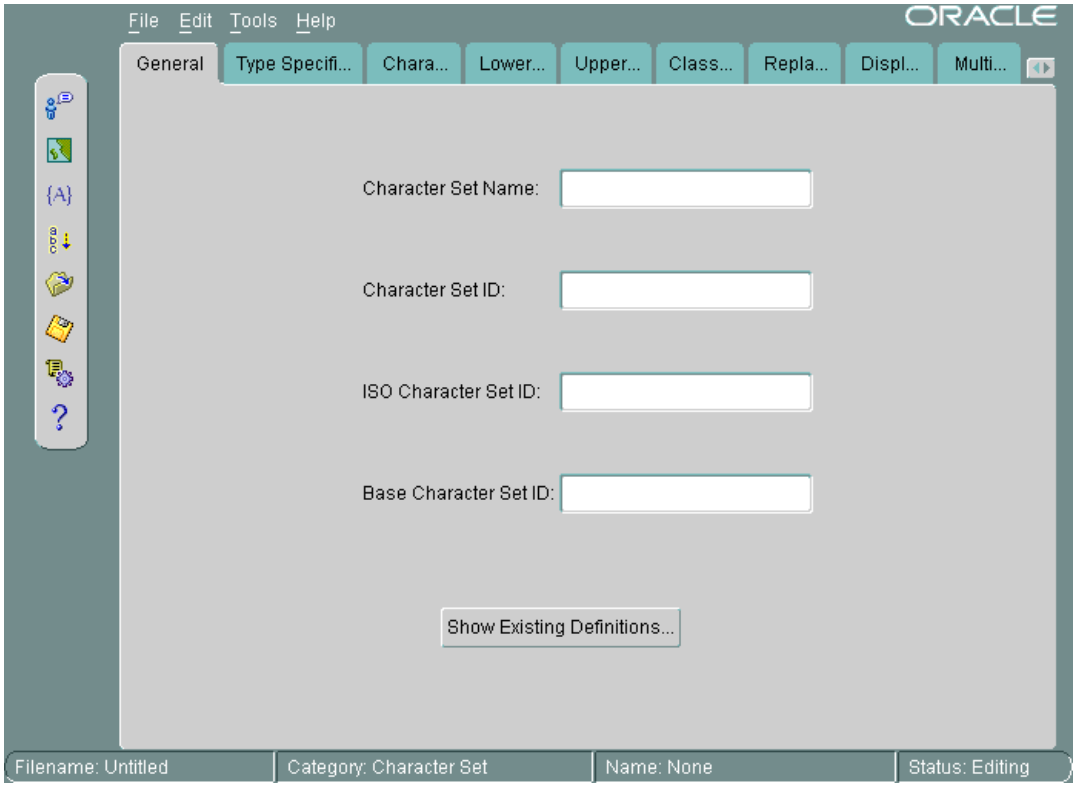


You can display the common languages and time zones for the current territory. For example, with a territory of CANADA, the common languages are ENGLISH, CANADIAN FRENCH, and FRENCH. The common time zones are America/Montreal, America/St_Johns, America/Halifax, America/Winnipeg, America/Regina, America/Edmonton, and America/Vancouver.

Displaying a Code Chart with the Oracle Locale Builder

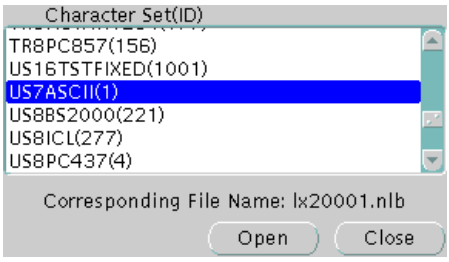
You can display and print the code charts of character sets with the Oracle Locale Builder. From the opening screen for Oracle Locale Builder, choose **File** > **New** > **Character Set**. Figure 12–16 shows the resulting screen.

Figure 12–16 General Tab Page for Character Sets



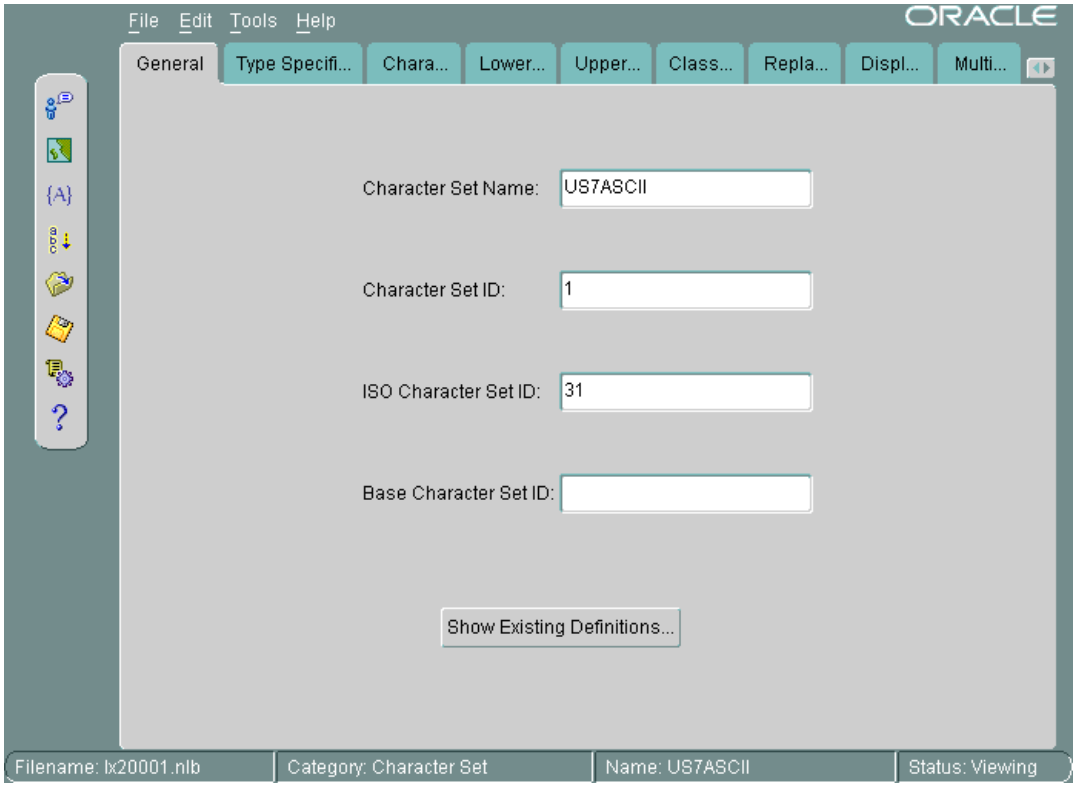
Click **Show Existing Definitions**. Highlight the character set you want to display. Figure 12–17 shows the Existing Definitions combo box with US7ASCII highlighted.

Figure 12–17 Choosing US7ASCII in the Existing Definitions Dialog Box



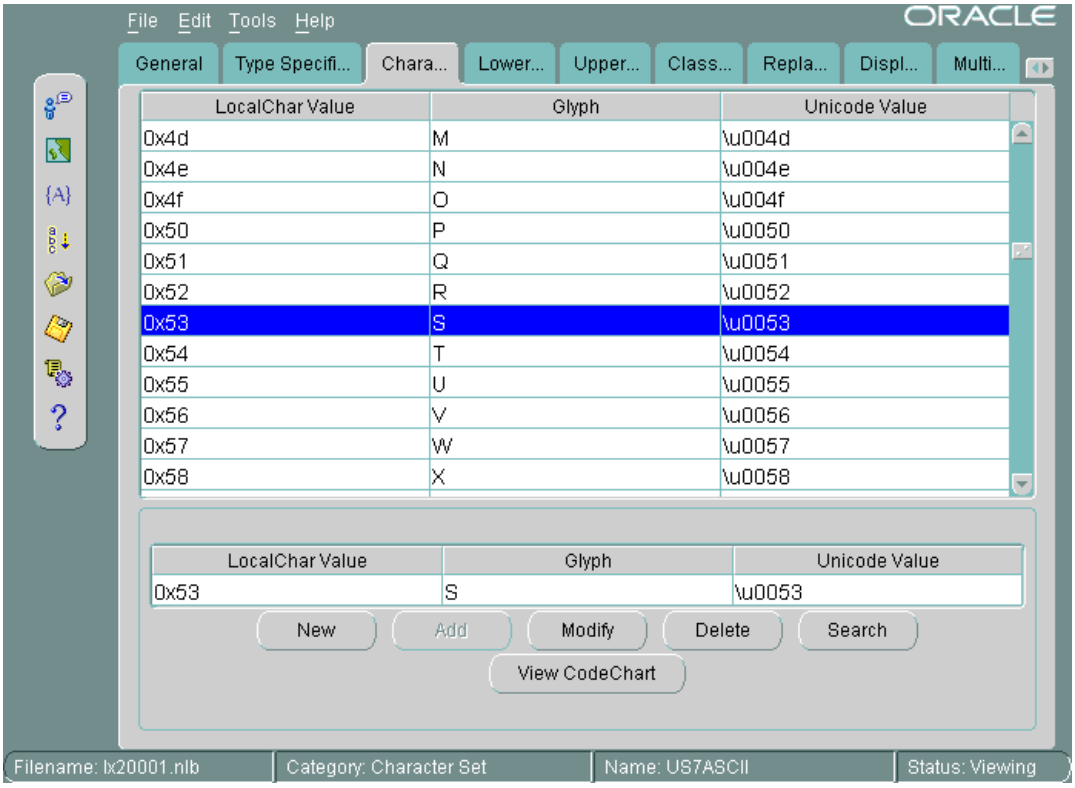
Click **Open** to choose the character set. Figure 12–18 shows the **General** tab page when US7ASCII has been chosen.

Figure 12–18 General Tab Page When US7ASCII Has Been Chosen



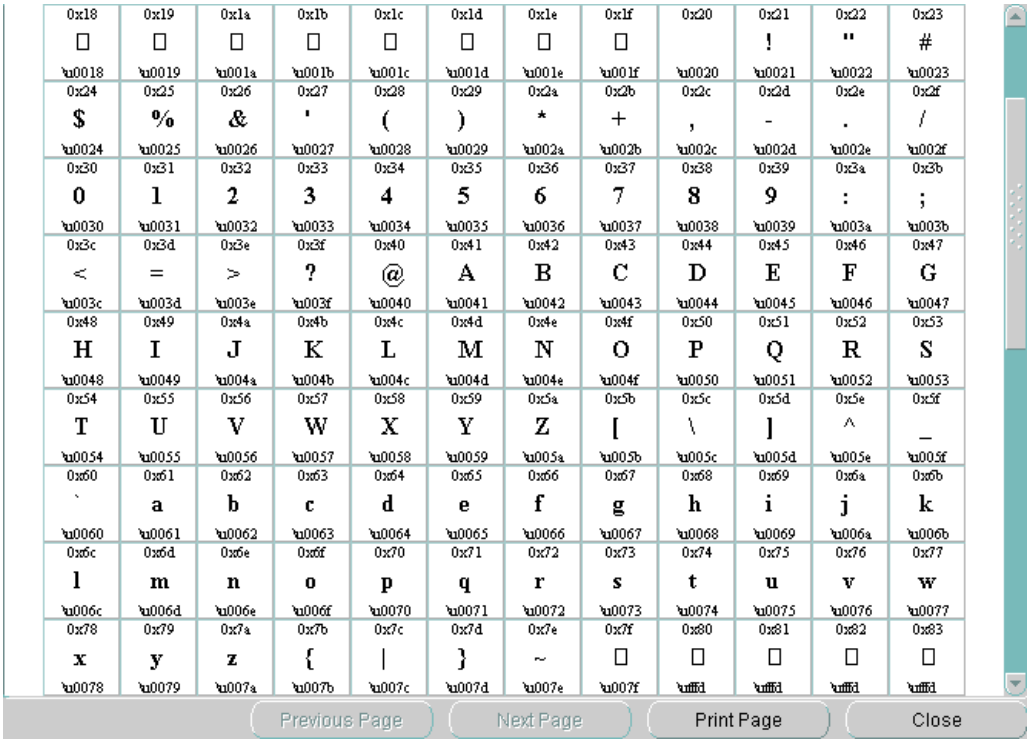
Click the **Character Data Mapping** tab. Figure 12–19 shows the **Character Data Mapping** tab page for US7ASCII.

Figure 12–19 Character Data Mapping Tab Page for US7ASCII



Click **View CodeChart**. Figure 12–20 shows the code chart for US7ASCII.

Figure 12–20 US7ASCII Code Chart



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. You can extend an existing encoded character set definition. User-defined characters are often used to encode special characters that represent the following language elements:

- 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 Database supports user-defined characters. It includes the following topics:

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
F940-F97E, F980-F9FC	188

The Oracle Database character sets listed in Table 12–2 contain predefined ranges that support user-defined characters.

Table 12–2 Oracle Database 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 Database 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 12–21.

Figure 12–21 Japanese Kanji Character

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

Oracle Database defines all character sets with respect to Unicode 6.2 code points. That is, each character is defined as a Unicode 6.2 code value. Character conversion takes place transparently by using Unicode as the intermediate form. For example, when a JA16SJIS client connects to a JA16EUC database, the character shown in Figure 12–21 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 JA16EUC (code point value B0A1).

Unicode 6.2 Private Use Area

Unicode 6.2 reserves the range E000-F8FF for the Private Use Area (PUA). The PUA is intended for end users' or vendors' private use character definition.

User-defined characters can be converted between two Oracle Database character sets by using Unicode 6.2 PUA as the intermediate form, which is the same as for standard characters.

User-Defined Character Cross-References Between Character Sets

Cross-references between different character sets are required when registering user-defined characters across operating systems. Cross-references ensure that the user-defined characters can be converted correctly across the different character sets when they are mapped to a Unicode PUA value.

For example, when registering a user-defined character on both a Japanese Shift-JIS operating system and a Japanese IBM Host operating system, you may want to assign the F040 code point on the Shift-JIS operating system and the 6941 code point on the IBM Host operating system for this character. This is so that Oracle Database can map this character correctly between the character sets JA16SJIS and JA16DBCS.

User-defined character cross-reference information can be found by viewing the character set definitions using the Oracle Locale Builder. For example, you can determine that both the Shift-JIS UDC value F040 and the IBM Host UDC value 6941 are mapped to the same Unicode PUA value E000.

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 ID for you. You can also choose your own character set ID. Use the following format for naming character set definition NLT files:

`lx2`*`dddd`*`.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 6.2 private use range is e000-f8ff. However, Oracle Database reserves f500-f8ff for its own private use.)
- No line in the character set definition file can be longer than 80 characters.

Note: When you create a new multibyte character set from an existing character set, use an 8-bit or multibyte character set as the original character set.

If you derive a new character set from an existing Oracle Database character set, then Oracle 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, then 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 Database

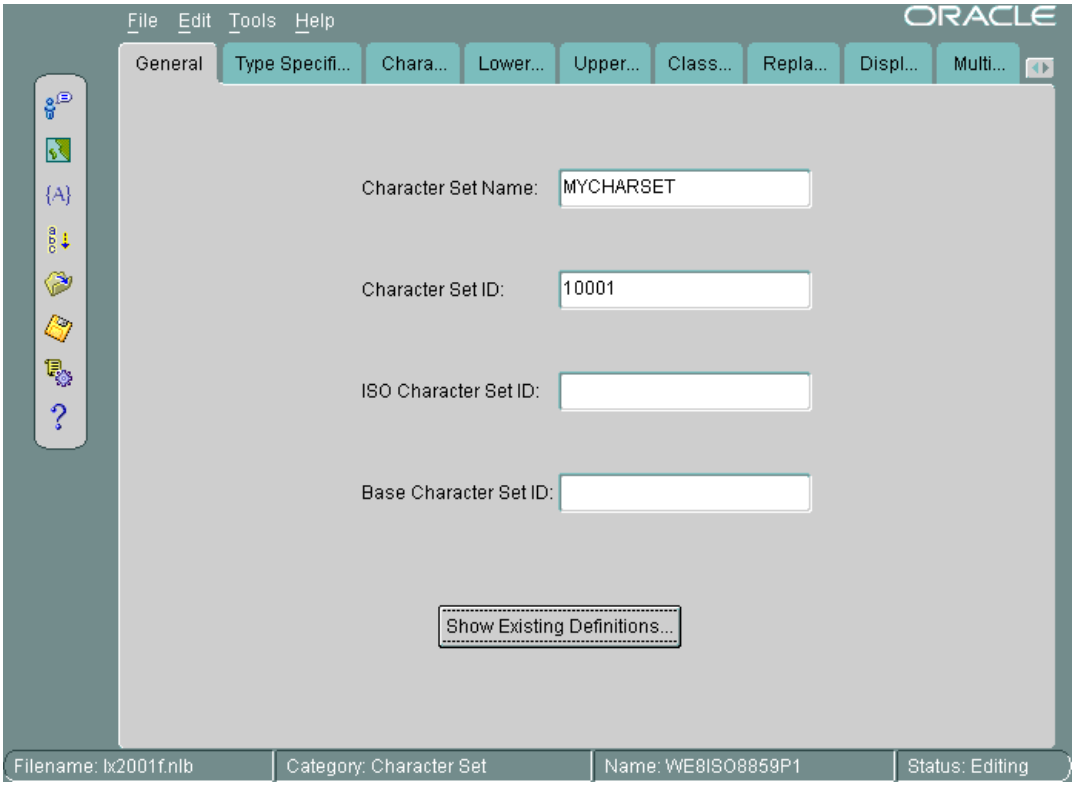
- SUNW represents the organization name (company stock trading abbreviation for Sun Microsystems)
- EXT specifies that this character set is an extension to the JA16EUC character set
- 1 specifies the version

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 uses the WE8ISO8859P1 character set and adds 10 Chinese characters.

Figure 12–22 shows the **General** tab page for MYCHARSET.

Figure 12–22 General Tab Page for MYCHARSET



Click **Show Existing Definitions** and choose the WE8ISO8859P1 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 8000 to 8999 or 10000 to 20000.

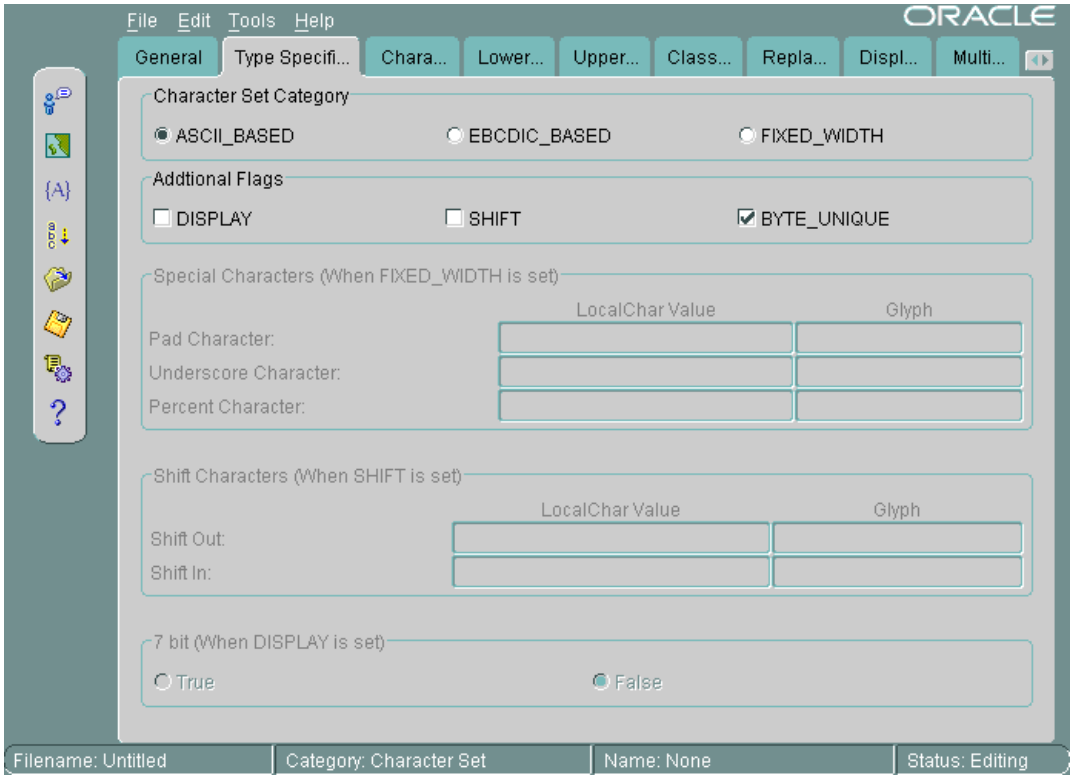
Note: If you are using Pro*COBOL, then choose a character set ID between 8000 and 8999.

The **ISO Character Set ID** field remains blank for user-defined character sets.

In this example, the **Base Character Set ID** field remains blank. However, you can specify a character set to use as a template. The settings in the **Type Specification** tab page must match the type settings of the base character set that you enter in the **Base Character Set ID** field. If the type settings do not match, then you will receive an error when you generate your custom character set.

Figure 12–23 shows the **Type Specification** tab page.

Figure 12–23 Type Specification Tab Page



The **Character Set Category** is `ASCII_BASED`. The **BYTE_UNIQUE** button is checked.

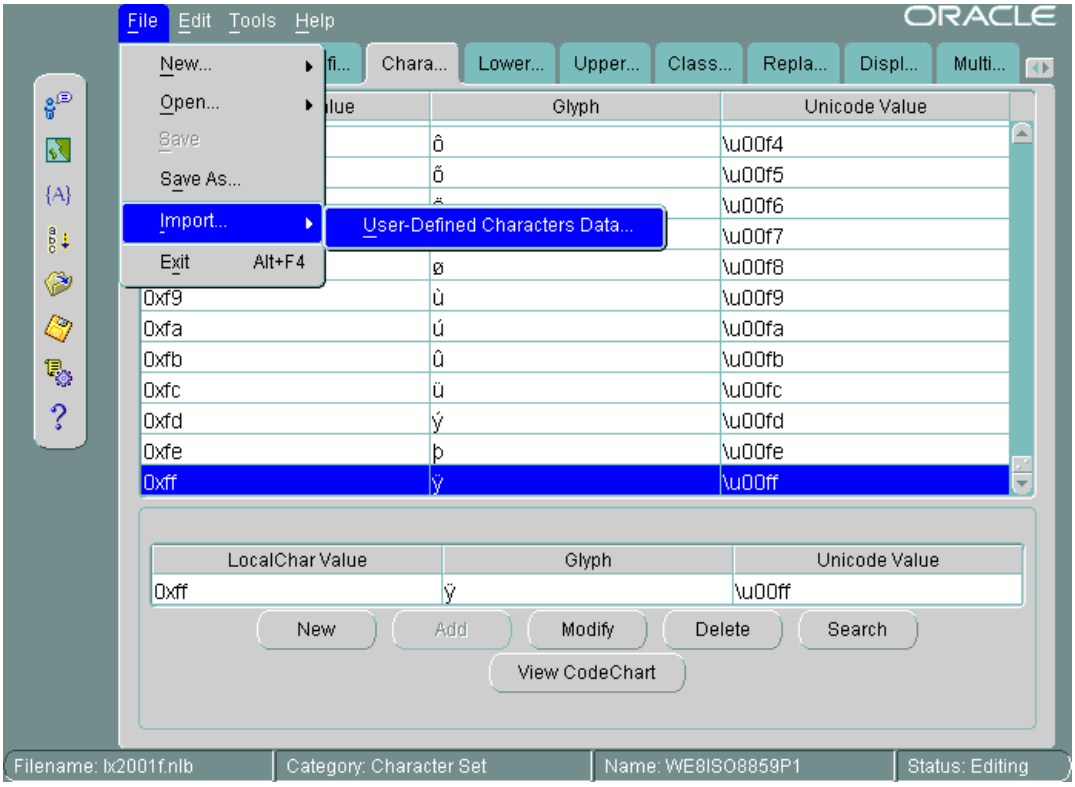
When you have chosen an existing character set, the fields for the **Type Specification** tab page should already be set to appropriate values. You should keep these values unless you have a specific reason for changing them. If you need to change the settings, then use the following guidelines:

- **FIXED_WIDTH** is used to identify character sets whose characters have a uniform length.
- **BYTE_UNIQUE** means that 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 used 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 2-7 for more information about shift-in and shift-out character sets

Figure 12–24 shows how to add user-defined characters.

Figure 12–24 Importing User-Defined Character Data



Open the **Character Data Mapping** tab page. Highlight the character that you want to add characters after in the character set. In this example, the `0xff` 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

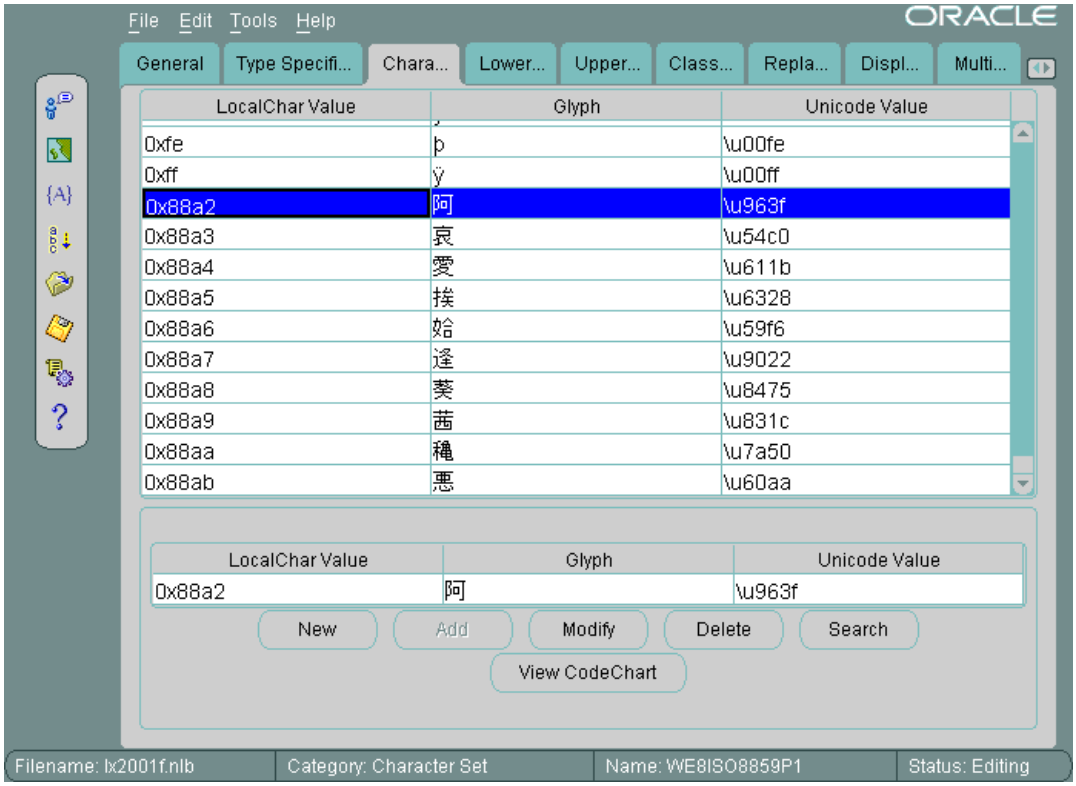
88aa 7a50

88ab 60aa

Choose **File** > **Import** > **User-Defined Characters Data**.

Figure 12–25 shows that the imported characters are added after `0xff` in the character set.

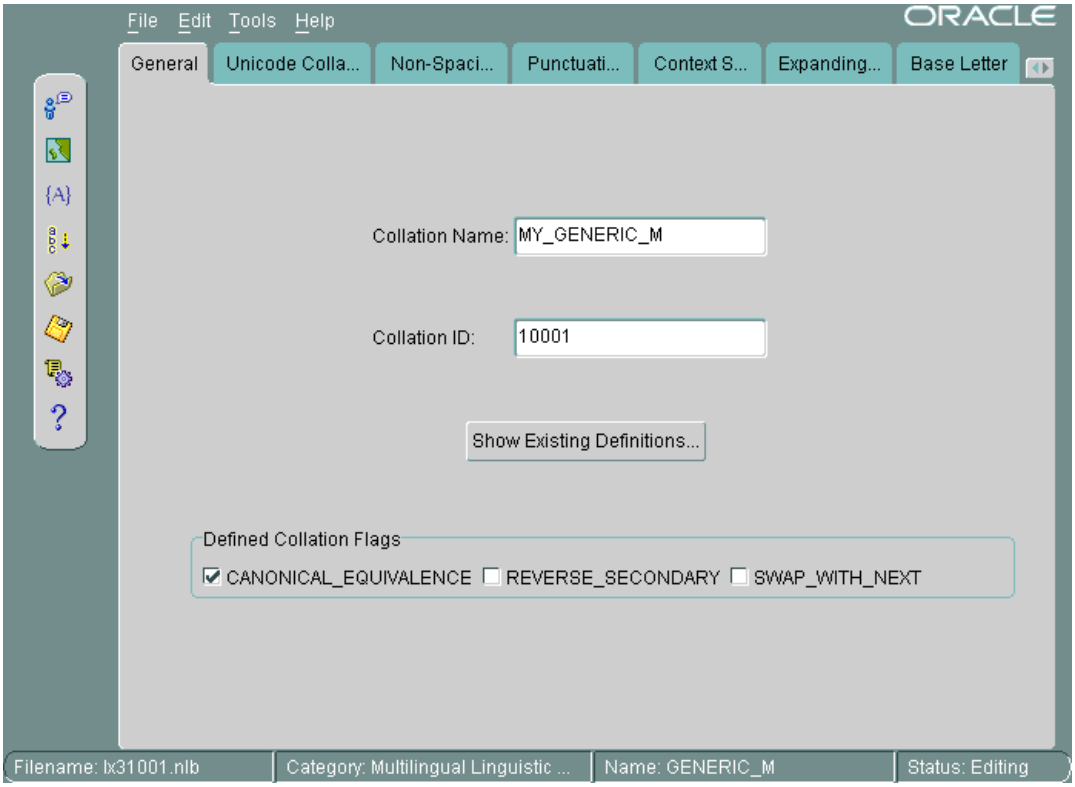
Figure 12–25 New Characters in the Character Set



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 sort is used as the basis for the new linguistic sort. Figure 12–26 shows how to begin.

Figure 12–26 General Tab Page for Collation



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 are used. In this example, **CANONICAL_EQUIVALENCE** is checked.

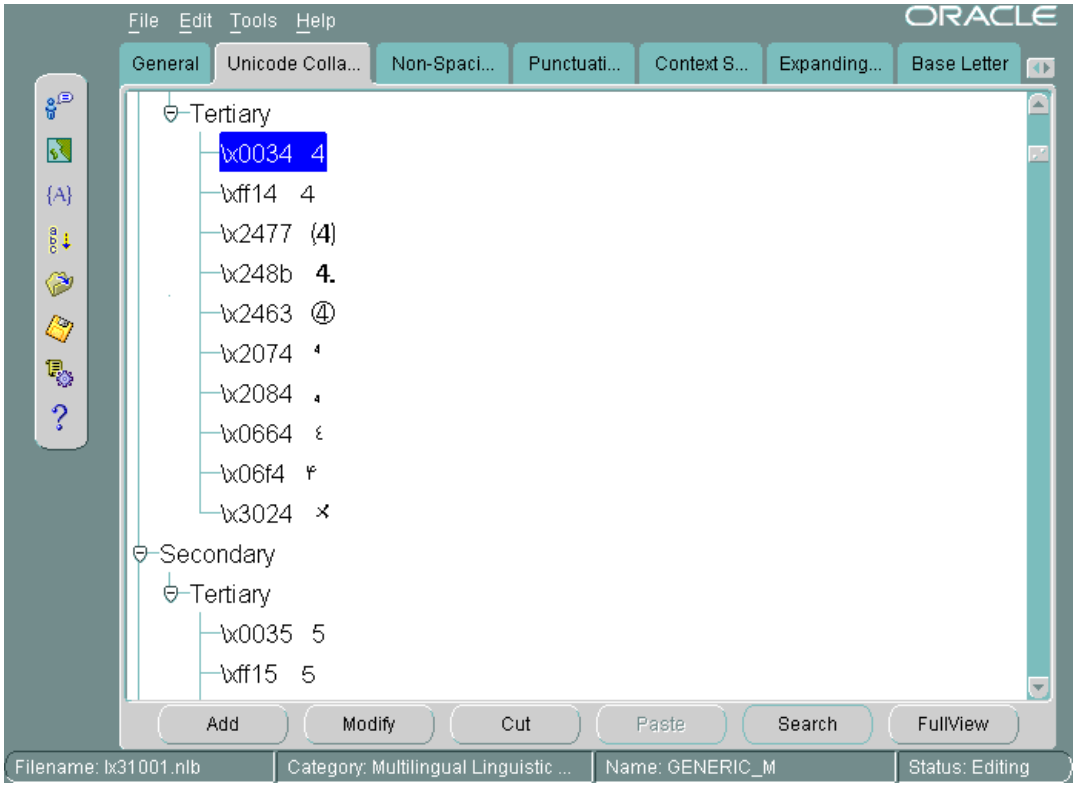
The valid range for **Collation ID** (sort ID) for a user-defined sort is 1000 to 2000 for monolingual collation and 10000 to 11000 for multilingual collation.

See Also:

- Figure 12–30, "Canonical Rules Dialog Box" for more information about canonical rules
- Chapter 5, "Linguistic Sorting and Matching"

Figure 12–27 shows the **Unicode Collation Sequence** tab page.

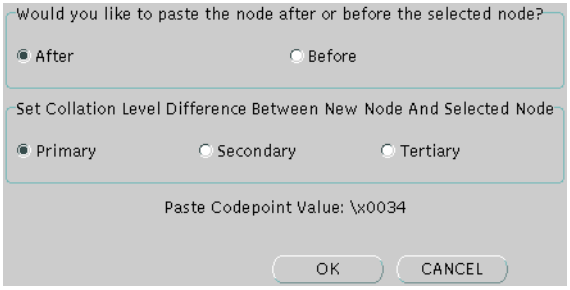
Figure 12–27 Unicode Collation Sequence Tab Page



This example customizes the linguistic sort 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 12–27, 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 12–28.

Figure 12–28 Paste Node Dialog Box



- 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.

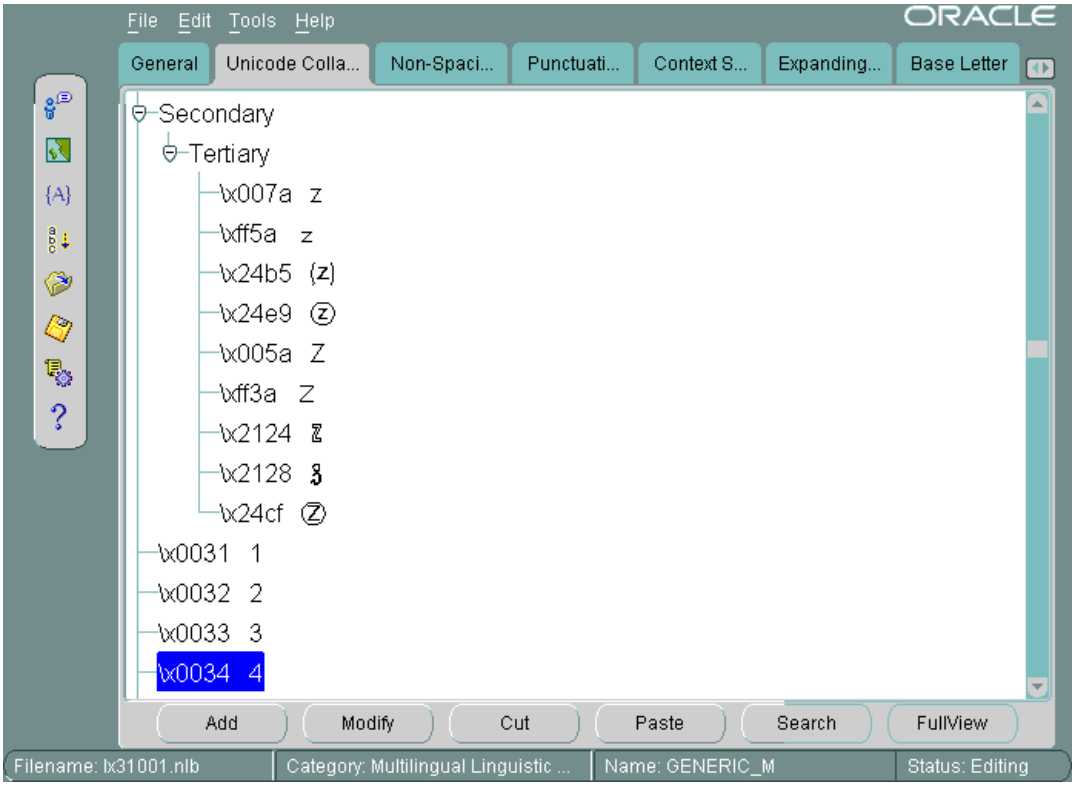
In Figure 12–28, the **After** button and the **Primary** button are selected.

- Click **OK** to paste the node.

  Use similar steps to move other digits to a position after the letters `a` through `z`.

Figure 12–29 shows the resulting **Unicode Collation Sequence** tab page after the digits `0` through `4` have been moved to a position after the letters `a` through `z`.

Figure 12–29 Unicode Collation Sequence After Modification



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 each character with a circumflex (for example, û) to be after the same character containing a tilde.

Verify the current sort order by choosing **Tools** > **Canonical Rules**. This opens the Canonical Rules dialog box, shown in Figure 12–30.

Figure 12–30 Canonical Rules Dialog Box

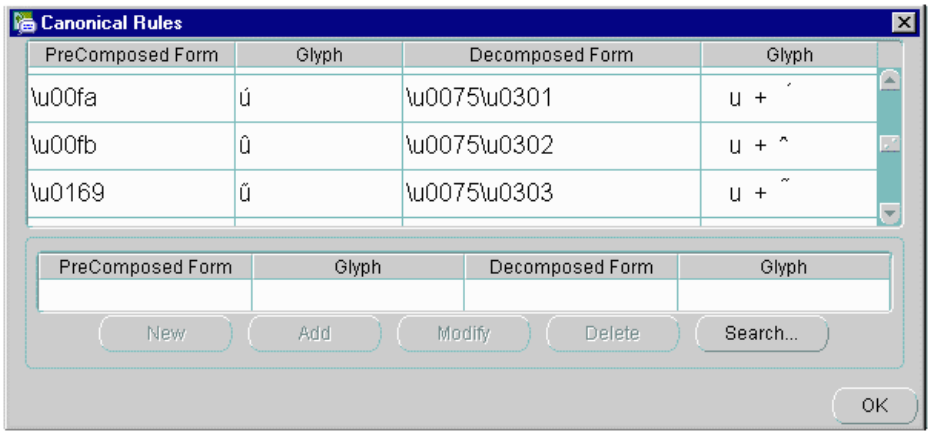
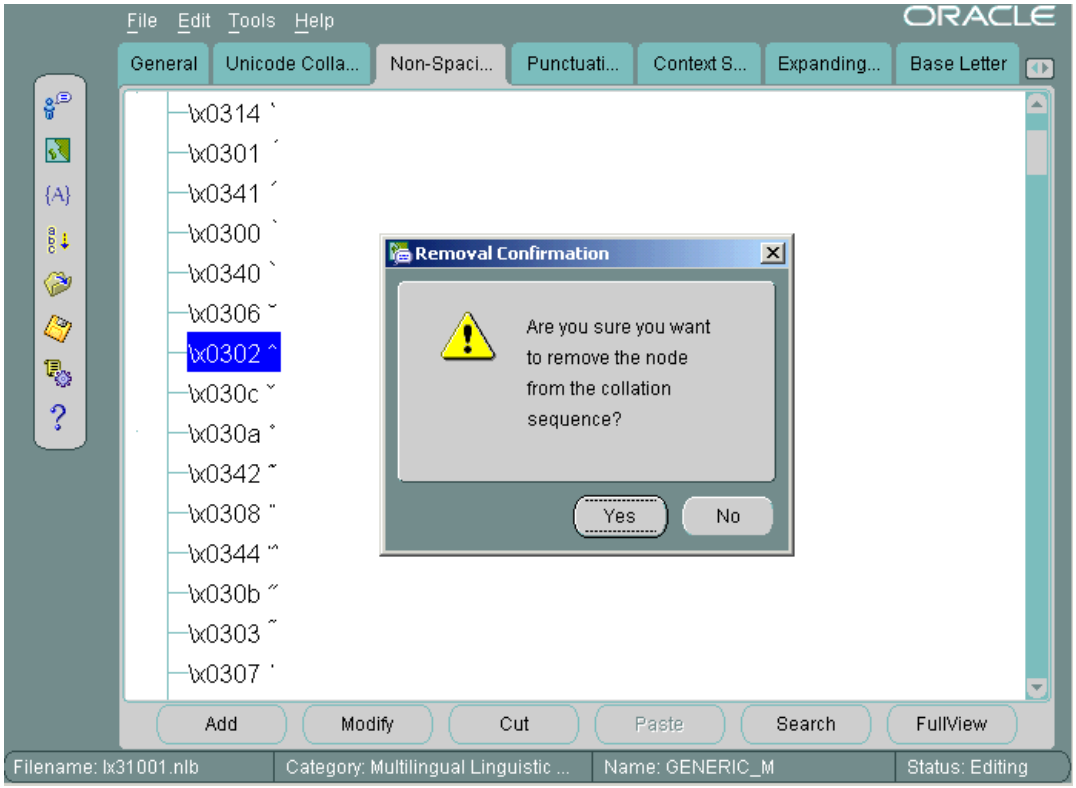


Figure 12–30 shows how characters are decomposed into their canonical equivalents and their current sorting orders. For example, û is represented as u plus ^.

> **See Also:** Chapter 5, "Linguistic Sorting and Matching" for more information about canonical rules

In the Oracle Locale Builder collation window (shown in Figure 12–26), click the **Non-Spacing Characters** tab. If you use the **Non-Spacing Characters** tab page, then changes for diacritics apply to all characters. Figure 12–31 shows the **Non-Spacing Characters** tab page.

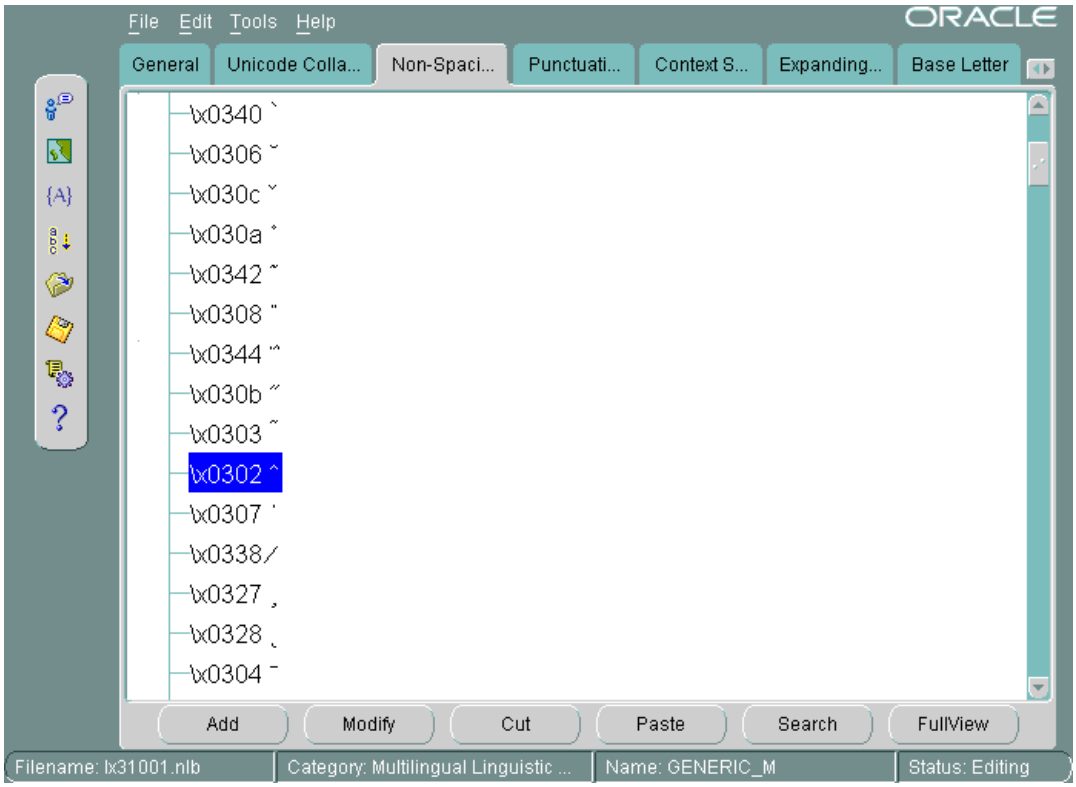
Figure 12–31 Changing the Sort Order for All Characters with the Same Diacritic



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 12–32 illustrates the new sort order.

Figure 12–32 The New Sort Order for Characters with the Same Diacritic



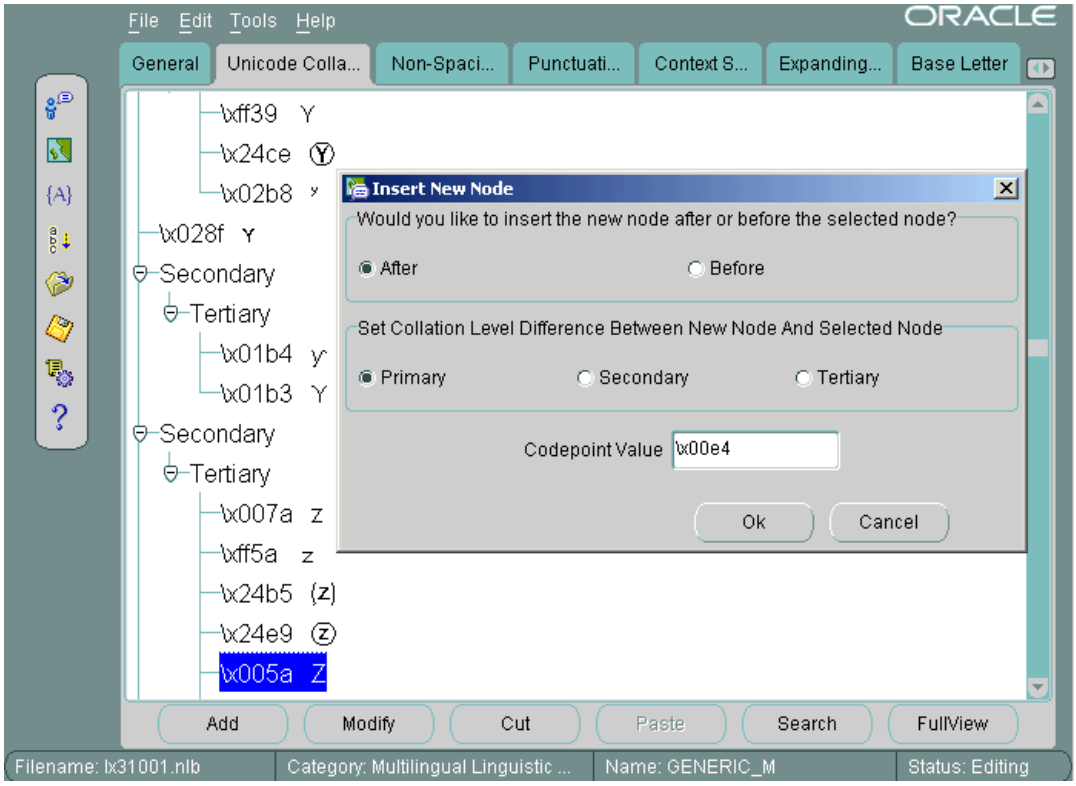
Changing the Sort Order for One Character with a Diacritic

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 Sequence** tab page, 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 12–33.

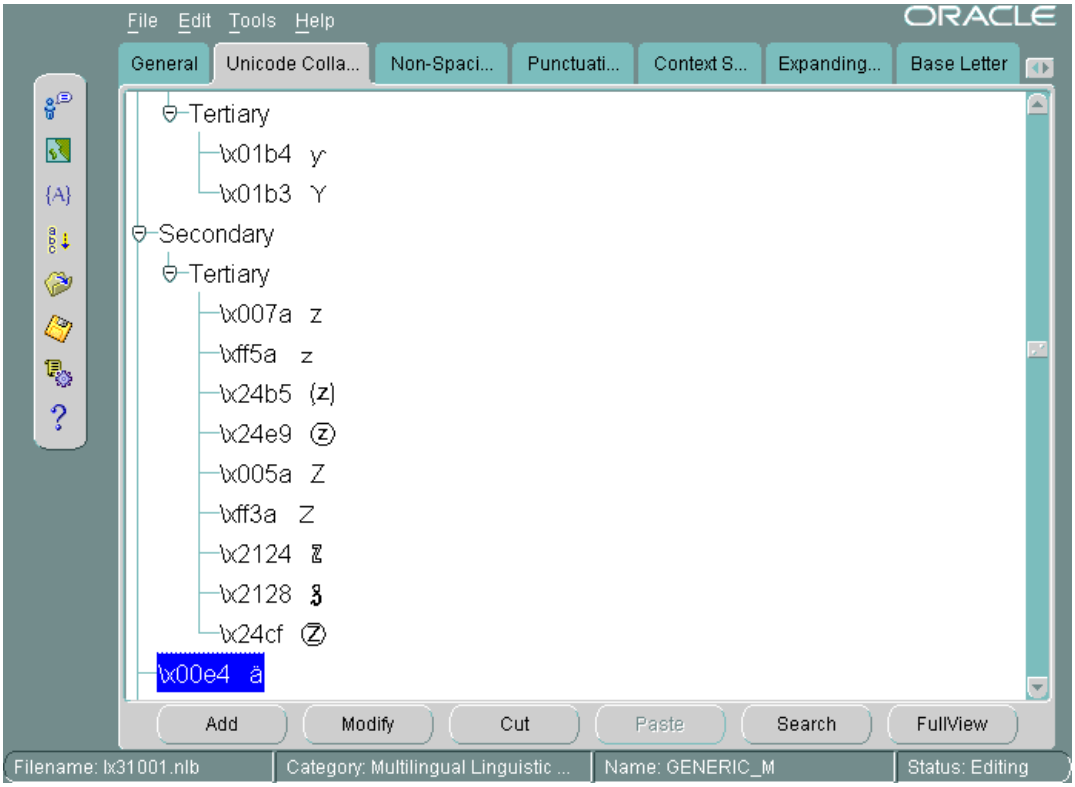
Figure 12–33 Changing the Sort Order of One Character with a Diacritic



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 12–34 shows the resulting sort order.

Figure 12–34 New Sort Order After Changing a Single Character



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 as follows.

1. As the user who owns the files (typically user `oracle`), back up the NLS installation boot file (`lx0boot.nlb`) and the NLS system boot file (`lx1boot.nlb`) in the `ORA_NLS10` directory. On a UNIX platform, enter commands similar to the following example:

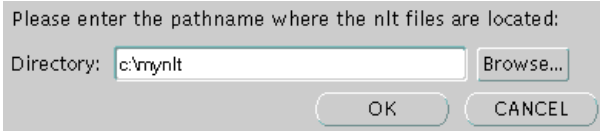
   ```
   % setenv ORA_NLS10 $ORACLE_HOME/nls/data
   % cd $ORA_NLS10
   % cp -p lx0boot.nlb lx0boot.nlb.orig
   % cp -p lx1boot.nlb lx1boot.nlb.orig
   ```

   Note that the `-p` option preserves the timestamp of the original file.

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 12–35.

Figure 12–35 Location Dialog Box

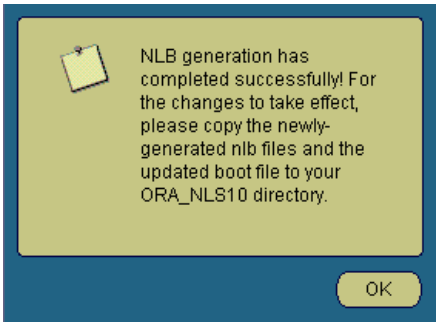


Do not try to specify an NLT file. Oracle Locale Builder generates an NLB file for each NLT file.

4. Click **OK** to generate the NLB files.

   Figure 12–36 illustrates the final notification that you have successfully generated NLB files for all NLT files in the directory.

Figure 12–36 NLB Generation Success Dialog Box



5. Copy the `lx1boot.nlb` file into the path that is specified by the `ORA_NLS10` environment variable. For example, on a UNIX platform, enter a command similar to the following example:

```
% cp /directory_name/lx1boot.nlb $ORA_NLS10/lx1boot.nlb
```

6. Copy the new NLB files into the `ORA_NLS10` directory. For example, on a UNIX platform, enter commands similar to the following example:

```
% cp /directory_name/lx22710.nlb $ORA_NLS10
% cp /directory_name/lx52710.nlb $ORA_NLS10
```

> **Note:** Oracle Locale Builder generates NLB files in the directory where the NLT files reside

7. Restart the database to use the newly created locale data.
8. To use the new locale data on the client side, exit the client and re-invoke the client after installing the NLB files.

> **See Also:** "Locale Data on Demand" on page 1-1 for more information about the `ORA_NLS10` environment variable

Upgrading Custom NLB Files from Previous Releases of Oracle Database

Locale definition files are database release-dependent. For example, NLB files from Oracle Database 9*i* and Oracle Database 10*g* are not directly supported in an Oracle Database 11 installation, and so forth. Even a patch set may introduce a small change to the NLB file format, if it is necessary to fix a bug. Installation of a patch set or a patch set update (PSU) may overwrite your customizations, if any Oracle-supplied NLB files have been modified in the patch set.

In order to migrate your locale customization files from your current release of the database to a new release or patch set, perform the following steps:

1. Create a directory and copy your customized NLB or NLT files there.

2. Install the new database release, patch set or patch set update into the existing Oracle Home or into a new Oracle Home, as appropriate.
3. Use the Oracle Locale Builder from the new or updated Oracle Home to open each of the files copied in step (1) and save them in the NLT format to the source directory.
4. Repeat the NLB generation and installation steps as described in the section "Generating and Installing NLB Files" on page 12-32, still using the new version of the Oracle Locale Builder and the same source directory.

Note that Oracle Locale Builder can read and process previous versions of the NLT and NLB files, as well as read and process these files from different platforms. However, Oracle Locale Builder always saves NLT files and generates NLB files in the latest format for the release of Oracle Database that you have installed.

Deploying Custom NLB Files to Oracle Installations on the Same Platform

To add your customizations to another Oracle Home with exactly the same database release and patch configuration and on the same platform as the Oracle Home used to generate the original set of customizations, perform the following steps:

1. In the target Oracle Home, perform step 1 from "Generating and Installing NLB Files" on page 12-32.
2. If the target Oracle Home is on another machine, copy your customized NLB files and the generated `lx1boot.nlb` file to the target machine, using any method preserving the binary integrity of the files, such as FTP in binary mode, copy to a remotely mounted file system, `rcp` utility, and so on.
3. On the target machine, perform steps 5-8 from "Generating and Installing NLB Files" on page 12-32 using the directory containing your customized NLB files and the `lx1boot.nlb` file as `directory_name`.

Deploying Custom NLB Files to Oracle Installations on Another Platform

While being release-dependent, NLB files are platform-independent. Platform-dependent differences in the binary format (such as 32-bit versus 64-bit, big-endian versus little-endian, ASCII versus EBCDIC) are processed transparently during NLB loading. Therefore, when deploying your locale customization files into other Oracle Database installations running with the same Oracle Database release and patch configuration, but under a different operating system platform, you can choose one of the following two options:

1. Copy over the custom `.NLT` files to your new platform and repeat the NLB generation and installation steps as described in "Generating and Installing NLB Files" on page 12-32.
2. Copy over the entire set of `.NLB` files (both Oracle-supplied NLB files and custom NLB files) to your new platform.

Note that option (2) may introduce some overhead at NLB loading time due to the transparent platform processing required. However, this overhead should be negligible, because each NLB file is usually loaded only once after an Oracle Database instance or an Oracle Client application is started and it is cached until the instance or application is shut down. Moreover, NLB files are loaded on demand. So, in most installations, only a small subset of all available NLB files is ever loaded into memory.

Option (2) is especially useful to customize files for platforms on which Oracle Locale Builder is not supported.

To copy over the entire set of NLB files to your new platform, perform the following steps:

1. Shut down all Oracle Database instances and Oracle Client applications using the target Oracle Home.
2. As the user who owns the files (typically user `oracle`), move all NLB files from the `ORA_NLS10` directory of the target Oracle Home to a backup directory. On a UNIX platform, enter commands similar to the following example:

```
% setenv ORA_NLS10 $ORACLE_HOME/nls/data
% cd $ORA_NLS10
% mkdir orig
% mv *.nlb orig
```

3. Copy all NLB files from the source Oracle Home NLB directory to the target Oracle Home NLB (`$ORA_NLS10`) directory. Use any remote copy method preserving the binary integrity of the files, such as FTP in binary mode, copy to a remotely mounted file system, `rcp` utility, and so on.
4. Restart the database instances and/or applications, as desired.

Adding Custom Locale Definitions to Java Components with the GINSTALL Utility

The `ginstall` utility adds custom character sets, language, territory, and linguistic sorts to Java components in your applications. You use Locale Builder to define your custom character sets, language, territory, and linguistic sort. Locale Builder generates NLT files, which contain the custom definitions. Then to add the custom definitions to the Java components, you run `ginstall` to generate `gdk_custom.jar`. The same procedures can be used for Oracle Database release 10.2 and 10.1, as well as release 11.1 and 11.2.

To add custom definitions for character set, language, territory, and linguistic sort:

1. Generate the NLT file using Oracle Locale Builder.

   If you are upgrading custom NLB files from a previous release, follow the procedure described in "Upgrading Custom NLB Files from Previous Releases of Oracle Database" on page 12-33.

2. Run `ginstall` with `-add` or `-a` option to generate `gdk_custom.jar`.

```
ginstall -[add | a] lx2dddd.nlt
```

   To generate multiple NLT files:

```
ginstall -[add | a] lx2ddd.nlt lx2dddd.nlt lx2dddd.nlt
```

3. Copy `gdk_custom.jar` to the same directory as `orai18n.jar` or `orai18n-mapping.jar`.

To remove a custom definition:

- Run `ginstall` as follows.

```
ginstall -[remove | r] <path to gdk_custom.jar> <name of NLT file>
```

To update a custom definition:

- Run `ginstall` as follows.

```
ginstall -[update | u] <path to gdk_custom.jar> <name of NLT file>
```

Customizing Calendars with the NLS Calendar Utility

Oracle Database supports several calendars. Some of them may require the addition of ruler eras in the future and some may require tailoring to local needs through addition or subtraction of deviation days. To add the required information to your Oracle implementation, you can use external files that are 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 for the `lxegen` utility are hard-coded and depend on the platform. On UNIX platforms, the file name is `lxecal.nlt`. It is located in the `$ORACLE_HOME/nls` directory. A sample text definition file is included in the `$ORACLE_HOME/nls/demo` directory. See *Oracle Database Examples Installation Guide* for more information regarding how to install demo files.

Depending on the number of different calendars referenced in the text definition file, the `lxegen` utility produces one or more binary files. The names of the binary files are also hard-coded and depend on the platform. On UNIX platforms, the names of the binary files are `lxecalah.nlb` (deviation days for the Arabic Hijrah calendar), `lxecaleh.nlb` (deviation days for the English Hijrah calendar), and `lxecalji.nlb` (ruler eras for the Japanese Imperial calendar). The binary files are generated in the same directory as the text file and overwrite existing binary files.

After the binary files have been generated, they are automatically loaded during system initialization. Do not move or rename the files. Unlike files generated by Oracle Locale Builder, calendar customization binary files are not platform-independent. You should generate them for each combination of Oracle software release and platform separately.

Invoke the calendar utility from the command line as follows:

```
% lxegen
```

See Also:

- Operating system documentation for the location of the files on your system
- "Calendar Systems" on page A-24

A

Locale Data

This appendix lists the languages, territories, character sets, and other locale data supported by Oracle Database. This appendix includes these topics:

- Languages
- Translated Messages
- Territories
- Character Sets
- Language and Character Set Detection Support
- Linguistic Sorts
- Calendar Systems
- Time Zone Region Names
- Obsolete Locale Data

You can obtain information about character sets, languages, territories, and linguistic sorts by querying the `V$NLS_VALID_VALUES` dynamic performance view.

> **See Also:** *Oracle Database Reference* for more information about the data that can be returned by this view

Languages

Languages in Table A–1 provide support for locale-sensitive information such as:

- Day and month names and their abbreviations
- Symbols for equivalent expressions for A.M., P.M., A.D., and B.C.
- Default sorting sequence for character data when the `ORDER BY` SQL clause is specified
- Writing direction (left to right or right to left)
- Affirmative and negative response strings (for example, `YES` and `NO`)

By using Unicode databases and data types, you can store, process, and retrieve data for almost all contemporary languages, including many that do not appear in Table A–1.

Table A–1 Oracle Database Supported Languages

Language Name	Language Abbreviation	Default Sort
ALBANIAN	sq	GENERIC_M
AMERICAN	us	binary
AMHARIC	am	GENERIC_M
ARABIC	ar	ARABIC
ARMENIAN	hy	GENERIC_M
ASSAMESE	as	binary
AZERBAIJANI	az	AZERBAIJANI
BANGLA	bn	binary
BELARUSIAN	be	RUSSIAN
BRAZILIAN PORTUGUESE	ptb	WEST_EUROPEAN
BULGARIAN	bg	BULGARIAN
CANADIAN FRENCH	frc	CANADIAN FRENCH
CATALAN	ca	CATALAN
CROATIAN	hr	CROATIAN
CYRILLIC KAZAKH	ckk	GENERIC_M
CYRILLIC SERBIAN	csr	GENERIC_M
CYRILLIC UZBEK	cuz	GENERIC_M
CZECH	cs	CZECH
DANISH	dk	DANISH
DARI	prs	GENERIC_M
DIVEHI	dv	GENERIC_M
DUTCH	nl	DUTCH
EGYPTIAN	eg	ARABIC
ENGLISH	gb	binary
ESTONIAN	et	ESTONIAN
FINNISH	sf	FINNISH
FRENCH	f	FRENCH
GERMAN DIN	din	GERMAN
GERMAN	d	GERMAN
GREEK	el	GREEK
GUJARATI	gu	binary
HEBREW	iw	HEBREW
HINDI	hi	binary
HUNGARIAN	hu	HUNGARIAN
ICELANDIC	is	ICELANDIC
INDONESIAN	in	INDONESIAN

Table A–1 (Cont.) Oracle Database Supported Languages

Language Name	Language Abbreviation	Default Sort
IRISH	ga	binary
ITALIAN	i	WEST_EUROPEAN
JAPANESE	ja	binary
KANNADA	kn	binary
KHMER	km	GENERIC_M
KOREAN	ko	binary
LAO	lo	GENERIC_M
LATIN AMERICAN SPANISH	esa	SPANISH
LATIN BOSNIAN	lbs	GENERIC_M
LATIN SERBIAN	lsr	binary
LATIN UZBEK	luz	GENERIC_M
LATVIAN	lv	LATVIAN
LITHUANIAN	lt	LITHUANIAN
MACEDONIAN	mk	binary
MALAY	ms	MALAY
MALAYALAM	ml	binary
MALTESE	mt	GENERIC_M
MARATHI	mr	binary
MEXICAN SPANISH	esm	WEST_EUROPEAN
NEPALI	ne	GENERIC_M
NORWEGIAN	n	NORWEGIAN
ORIYA	or	binary
PERSIAN	fa	GENERIC_M
POLISH	pl	POLISH
PORTUGUESE	pt	WEST_EUROPEAN
PUNJABI	pa	binary
ROMANIAN	ro	ROMANIAN
RUSSIAN	ru	RUSSIAN
SIMPLIFIED CHINESE	zhs	binary
SINHALA	si	GENERIC_M
SLOVAK	sk	SLOVAK
SLOVENIAN	sl	SLOVENIAN
SPANISH	e	SPANISH
SWAHILI	sw	GENERIC_M
SWEDISH	s	SWEDISH
TAMIL	ta	binary

Table A–1 (Cont.) Oracle Database Supported Languages

Language Name	Language Abbreviation	Default Sort
TELUGU	te	binary
THAI	th	THAI_DICTIONARY
TRADITIONAL CHINESE	zht	binary
TURKISH	tr	TURKISH
UKRAINIAN	uk	UKRAINIAN
VIETNAMESE	vn	VIETNAMESE

Translated Messages

Oracle Database error messages have been translated into the languages which are listed in Table A–2.

Table A–2 Oracle Database Supported Messages

Name	Abbreviation
ARABIC	ar
BRAZILIAN PORTUGUESE	ptb
CATALAN	ca
CZECH	cs
DANISH	dk
DUTCH	nl
FINNISH	sf
FRENCH	f
GERMAN	d
GREEK	el
HEBREW	iw
HUNGARIAN	hu
ITALIAN	i
JAPANESE	ja
KOREAN	ko
NORWEGIAN	n
POLISH	pl
PORTUGUESE	pt
ROMANIAN	ro
RUSSIAN	ru
SIMPLIFIED CHINESE	zhs
SLOVAK	sk
SPANISH	e
SWEDISH	s

Table A–2 (Cont.) Oracle Database Supported Messages

Name	Abbreviation
THAI	th
TRADITIONAL CHINESE	zht
TURKISH	tr

Territories

Table A–3 lists the territories that Oracle Database supports.

Table A–3 Oracle Database Supported Territories

Name	Name	Name
AFGHANISTAN	FYR MACEDONIA	PANAMA
ALBANIA	GABON	PARAGUAY
ALGERIA	GUATEMALA	PERU
AMERICA	GERMANY	PHILIPPINES
ARGENTINA	GREECE	POLAND
ARMENIA	HONDURAS	PORTUGAL
AUSTRALIA	HONG KONG	PUERTO RICO
AUSTRIA	HUNGARY	QATAR
AZERBAIJAN	ICELAND	ROMANIA
BAHAMAS	INDIA	RUSSIA
BAHRAIN	INDONESIA	SAUDI ARABIA
BANGLADESH	IRAQ	SENEGAL
BELARUS	IRAN	SERBIA
BELGIUM	IRELAND	SINGAPORE
BELIZE	ISRAEL	SLOVAKIA
BERMUDA	ITALY	SLOVENIA
BOLIVIA	IVORY COAST	SOMALIA
BOSNIA AND HERZEGOVINA	JAPAN	SOUTH AFRICA
BRAZIL	JORDAN	SPAIN
BULGARIA	KENYA	SRI LANKA
CAMBODIA	KOREA	SUDAN
CAMEROON	KUWAIT	SWEDEN
CATALONIA	LAOS	SWITZERLAND
CHILE	LATVIA	SYRIA
CHINA	LEBANON	TAIWAN
COLOMBIA	LIBYA	TANZANIA
CONGO BRAZZAVILLE	LITHUANIA	THAILAND

Table A–3 (Cont.) Oracle Database Supported Territories

Name	Name	Name
CONGO KINSHASA	LUXEMBOURG	THE NETHERLANDS
COSTA RICA	MALDIVES	TUNISIA
CROATIA	MALTA	TURKEY
CYPRUS	MAURITANIA	UGANDA
CZECH REPUBLIC	MEXICO	UKRAINE
DENMARK	MONTENEGRO	UNITED ARAB EMIRATES
DJIBOUTI	MOROCCO	UNITED KINGDOM
ECUADOR	NEW ZEALAND	URUGUAY
EGYPT	NEPAL	UZBEKISTAN
EL SALVADOR	NICARAGUA	VENEZUELA
ESTONIA	NIGERIA	VIETNAM
ETHIOPIA	NORWAY	YEMEN
FINLAND	OMAN	ZAMBIA
FRANCE	PAKISTAN	

Character Sets

The character sets that Oracle Database supports are listed in the following sections according to three broad categories.

- Recommended Database Character Sets
- Other Character Sets
- Client-Only Character Sets

In addition, common character set subset/superset combinations are listed. Some character sets can only be used with certain data types. For example, 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 example, the information includes whether the character set supports the euro currency symbol, whether user-defined characters are supported, and whether the character set is a strict superset of ASCII. (You can use the Database Migration Assistant for Unicode to migrate an existing database to a new character set, only if all of the schema data is a strict subset of the new character set.)

The key for the comment column of the character set tables is:

SB: single-byte encoding
MB: multibyte encoding
FIXED: fixed-width multibyte encoding
ASCII: strict superset of ASCII
EURO: euro symbol supported
UDC: user-defined characters supported

Oracle Database does not document individual code page layouts. For specific details about a particular character set, its character repertoire, and code point values, you can

use Oracle Locale Builder. Otherwise, you should refer to the actual national, international, or vendor-specific standards.

See Also:

- *Oracle Database Migration Assistant for Unicode Guide*
- Chapter 12, "Customizing Locale Data"

Recommended Database Character Sets

Table A–4 lists the recommended and most commonly used ASCII-based Oracle Database character sets. The list is ordered alphabetically within their respective language group.

Table A–4 Recommended ASCII Database Character Sets

	Name	Description	Comments
Asian	-	-	-
-	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
-	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
-	KO16MSWIN949	MS Windows Code Page 949 Korean	MB, ASCII, UDC
-	TH8TISASCII	Thai Industrial Standard 620-2533 - ASCII 8-bit	SB, ASCII, EURO
-	VN8MSWIN1258	MS Windows Code Page 1258 8-bit Vietnamese	SB, ASCII, EURO
-	ZHS16GBK	GBK 16-bit Simplified Chinese	MB, ASCII, UDC
-	ZHT16HKSCS	MS Windows Code Page 950 with Hong Kong Supplementary Character Set HKSCS-2001 (character set conversion to and from Unicode is based on Unicode 3.0)	MB, ASCII, EURO
-	ZHT16MSWIN950	MS Windows Code Page 950 Traditional Chinese	MB, ASCII, UDC
-	ZHT32EUC	EUC 32-bit Traditional Chinese	MB, ASCII
European	-	-	-
-	BLT8ISO8859P13	ISO 8859-13 Baltic	SB, ASCII
-	BLT8MSWIN1257	MS Windows Code Page 1257 8-bit Baltic	SB, ASCII, EURO
-	CL8ISO8859P5	ISO 8859-5 Latin/Cyrillic	SB, ASCII
-	CL8MSWIN1251	MS Windows Code Page 1251 8-bit Latin/Cyrillic	SB, ASCII, EURO
-	EE8ISO8859P2	ISO 8859-2 East European	SB, ASCII
-	EL8ISO8859P7	ISO 8859-7 Latin/Greek	SB, ASCII, EURO
-	EL8MSWIN1253	MS Windows Code Page 1253 8-bit Latin/Greek	SB, ASCII, EURO
-	EE8MSWIN1250	MS Windows Code Page 1250 8-bit East European	SB, ASCII, EURO
-	NE8ISO8859P10	ISO 8859-10 North European	SB, ASCII
-	NEE8ISO8859P4	ISO 8859-4 North and North-East European	SB, ASCII
-	WE8ISO8859P15	ISO 8859-15 West European	SB, ASCII, EURO
-	WE8MSWIN1252	MS Windows Code Page 1252 8-bit West European	SB, ASCII, EURO

Table A–4 (Cont.) Recommended ASCII Database Character Sets

	Name	Description	Comments
Middle Eastern	-	-	-
-	AR8ISO8859P6	ISO 8859-6 Latin/Arabic	SB, ASCII
-	AR8MSWIN1256	MS Windows Code Page 1256 8-Bit Latin/Arabic	SB, ASCII, EURO
-	IW8ISO8859P8	ISO 8859-8 Latin/Hebrew	SB, ASCII
-	IW8MSWIN1255	MS Windows Code Page 1255 8-bit Latin/Hebrew	SB, ASCII, EURO
-	TR8MSWIN1254	MS Windows Code Page 1254 8-bit Turkish	SB, ASCII, EURO
-	WE8ISO8859P9	ISO 8859-9 West European & Turkish	SB, ASCII
Universal	-	-	-
-	AL32UTF8	Unicode 6.2 UTF-8 Universal character set	MB, ASCII, EURO

Table A–5 lists the recommended and most commonly used EBCDIC-based Oracle Database character sets. The list is ordered alphabetically within their respective language group.

Table A–5 Recommended EBCDIC Database Character Sets

	Name	Description	Comments
Asian	-	-	-
-	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
-	TH8TISEBCDICS	Thai Industrial Standard 620-2533-EBCDIC Server 8-bit	SB
European	-	-	-
-	BLT8EBCDIC1112S	EBCDIC Code Page 1112 8-bit Server Baltic Multilingual	SB
-	CE8BS2000	Siemens EBCDIC.DF.04 8-bit Central European	SB
-	CL8BS2000	Siemens EBCDIC.EHC.LC 8-bit Cyrillic	SB
-	CL8EBCDIC1025R	EBCDIC Code Page 1025 Server 8-bit Cyrillic	SB
-	CL8EBCDIC1158R	EBCDIC Code Page 1158 Server 8-bit Cyrillic	SB
-	D8EBCDIC1141	EBCDIC Code Page 1141 8-bit Austrian German	SB, EURO
-	DK8EBCDIC1142	EBCDIC Code Page 1142 8-bit Danish	SB, EURO
-	EE8BS2000	Siemens EBCDIC.DF.04 8-bit East European	SB
-	EE8EBCDIC870S	EBCDIC Code Page 870 Server 8-bit East European	SB
-	EL8EBCDIC423R	IBM EBCDIC Code Page 423 for RDBMS server-side	SB
-	EL8EBCDIC875R	EBCDIC Code Page 875 Server 8-bit Greek	SB
-	F8EBCDIC1147	EBCDIC Code Page 1147 8-bit French	SB, EURO
-	I8EBCDIC1144	EBCDIC Code Page 1144 8-bit Italian	SB, EURO
-	SE8EBCDCI1143	EBCDIC Code Page 1143 8-bit Swedish	SB, EURO
-	WE8BS2000	Siemens EBCDIC.DF.04 8-bit West European	SB

Table A–5 (Cont.) Recommended EBCDIC Database Character Sets

	Name	Description	Comments
-	WE8BS2000E	Siemens EBCDIC.DF.04 8-bit West European	SB, EURO
-	WE8BS2000L5	Siemens EBCDIC.DF.L5 8-bit West European/Turkish	SB
-	WE8EBCDIC1047E	Latin 1/Open Systems 1047	SB, EBCDIC, EURO
-	WE8EBCDIC1140	EBCDIC Code Page 1140 8-bit West European	SB, EURO
-	WE8EBCDIC1145	EBCDIC Code Page 1145 8-bit West European	SB, EURO
-	WE8DBCDIC1146	EBCDIC Code Page 1146 8-bit West European	SB, EURO
-	WE8EBCDIC1148	EBCDIC Code Page 1148 8-bit West European	SB, EURO
Middle Eastern	-	-	-
-	AR8EBCDIC420S	EBCDIC Code Page 420 Server 8-bit Latin/Arabic	SB
-	IW8EBCDIC424S	EBCDIC Code Page 424 Server 8-bit Latin/Hebrew	SB
-	TR8EBCDIC1026S	EBCDIC Code Page 1026 Server 8-bit Turkish	SB

Other Character Sets

Table A–6 lists the other ASCII-based Oracle Database character sets. The list is ordered alphabetically within their language groups.

Table A–6 Other ASCII-based Database Character Sets

	Name	Description	Comments
Asian	-	-	-
-	BN8BSCII	Bangladesh National Code 8-bit BSCII	SB, ASCII
-	IN8ISCII	Multiple-Script Indian Standard 8-bit Latin/Indian Languages	SB, ASCII
-	JA16VMS	JVMS 16-bit Japanese	MB, ASCII
-	KO16KSC5601	KSC5601 16-bit Korean	MB, ASCII
-	KO16KSCCS	KSCCS 16-bit (Johab) Korean	MB, ASCII
-	TH8MACTHAIS	Mac Server 8-bit Latin/Thai	SB, ASCII
-	VN8VN3	VN3 8-bit Vietnamese	SB, ASCII
-	ZHS16CGB231280	CGB2312-80 16-bit Simplified Chinese	MB, ASCII
-	ZHT16BIG5	BIG5 16-bit Traditional Chinese	MB, ASCII
-	ZHT16CCDC	HP CCDC 16-bit Traditional Chinese	MB, ASCII
-	ZHT16DBT	Taiwan Taxation 16-bit Traditional Chinese	MB, ASCII
-	ZHT16HKSCS31	MS Windows Code Page 950 with Hong Kong Supplementary Character Set HKSCS-2001 (character set conversion to and from Unicode is based on Unicode 3.1)	MB, ASCII, EURO
-	ZHT32SOPS	SOPS 32-bit Traditional Chinese	MB, ASCII
-	ZHT32TRIS	TRIS 32-bit Traditional Chinese	MB, ASCII
Middle Eastern	-	-	-
-	AR8ADOS710	Arabic MS-DOS 710 Server 8-bit Latin/Arabic	SB, ASCII

Table A–6 (Cont.) Other ASCII-based Database Character Sets

	Name	Description	Comments
-	AR8ADOS720	Arabic MS-DOS 720 Server 8-bit Latin/Arabic	SB, ASCII
-	AR8APTEC715	APTEC 715 Server 8-bit Latin/Arabic	SB, ASCII
-	AR8ARABICMACS	Mac Server 8-bit Latin/Arabic	SB, ASCII
-	AR8ASMO8X	ASMO Extended 708 8-bit Latin/Arabic	SB, ASCII
-	AR8MUSSAD768	Mussa'd Alarabi/2 768 Server 8-bit Latin/Arabic	SB, ASCII
-	AR8NAFITHA711	Nafitha Enhanced 711 Server 8-bit Latin/Arabic	SB, ASCII
-	AR8NAFITHA721	Nafitha International 721 Server 8-bit Latin/Arabic	SB, ASCII
-	AR8SAKHR706	SAKHR 706 Server 8-bit Latin/Arabic	SB, ASCII
-	AR8SAKHR707	SAKHR 707 Server 8-bit Latin/Arabic	SB, ASCII
-	AZ8ISO8859P9E	ISO 8859-9 Latin Azerbaijani	SB, ASCII
-	IN8ISCII	Multiple-Script Indian Standard 8-bit Latin/Indian Languages	SB, ASCII
-	IW8MACHEBREWS	Mac Server 8-bit Hebrew	SB, ASCII
-	IW8PC1507	IBM-PC Code Page 1507/862 8-bit Latin/Hebrew	SB, ASCII
-	LA8ISO6937	ISO 6937 8-bit Coded Character Set for Text Communication	SB, ASCII
-	TR8DEC	DEC 8-bit Turkish	SB, ASCII
-	TR8MACTURKISHS	Mac Server 8-bit Turkish	SB, ASCII
-	TR8PC857	IBM-PC Code Page 857 8-bit Turkish	SB, ASCII
European	-	-	-
-	BG8MSWIN	MS Windows 8-bit Bulgarian Cyrillic	SB, ASCII
-	BG8PC437S	IBM-PC Code Page 437 8-bit (Bulgarian Modification)	SB, ASCII
-	BLT8CP921	Latvian Standard LVS8-92(1) Windows/Unix 8-bit Baltic	SB, ASCII
-	BLT8PC775	IBM-PC Code Page 775 8-bit Baltic	SB, ASCII
-	CDN8PC863	IBM-PC Code Page 863 8-bit Canadian French	SB, ASCII
-	CEL8ISO8859P14	ISO 8859-14 Celtic	SB, ASCII
-	CL8ISOIR111	ISOIR111 Cyrillic	SB, ASCII
-	CL8KOI8R	RELCOM Internet Standard 8-bit Latin/Cyrillic	SB, ASCII
-	CL8KOI8U	KOI8 Ukrainian Cyrillic	SB, ASCII
-	CL8MACCYRILLICS	Mac Server 8-bit Latin/Cyrillic	SB, ASCII
-	EE8MACCES	Mac Server 8-bit Central European	SB, ASCII
-	EE8MACCROATIANS	Mac Server 8-bit Croatian	SB, ASCII
-	EE8PC852	IBM-PC Code Page 852 8-bit East European	SB, ASCII
-	EL8DEC	DEC 8-bit Latin/Greek	SB, ASCII
-	EL8MACGREEKS	Mac Server 8-bit Greek	SB, ASCII
-	EL8PC437S	IBM-PC Code Page 437 8-bit (Greek modification)	SB, ASCII
-	EL8PC851	IBM-PC Code Page 851 8-bit Greek/Latin	SB, ASCII
-	EL8PC869	IBM-PC Code Page 869 8-bit Greek/Latin	SB, ASCII

Table A–6 (Cont.) Other ASCII-based Database Character Sets

	Name	Description	Comments
-	ET8MSWIN923	MS Windows Code Page 923 8-bit Estonian	SB, ASCII
-	HU8ABMOD	Hungarian 8-bit Special AB Mod	SB, ASCII
-	HU8CWI2	Hungarian 8-bit CWI-2	SB, ASCII
-	IS8PC861	IBM-PC Code Page 861 8-bit Icelandic	SB, ASCII
-	LA8ISO6937	ISO 6937 8-bit Coded Character Set for Text Communication	SB, ASCII
-	LA8PASSPORT	German Government Printer 8-bit All-European Latin	SB, ASCII
-	LT8MSWIN921	MS Windows Code Page 921 8-bit Lithuanian	SB, ASCII
-	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
-	LV8PC8LR	Latvian Version IBM-PC Code Page 866 8-bit Latin/Cyrillic	SB, ASCII
-	LV8PC1117	IBM-PC Code Page 1117 8-bit Latvian	SB, ASCII
-	LV8RST104090	IBM-PC Alternative Code Page 8-bit Latvian (Latin/Cyrillic)	SB, ASCII
-	N8PC865	IBM-PC Code Page 865 8-bit Norwegian	SB, ASCII
-	RU8BESTA	BESTA 8-bit Latin/Cyrillic	SB, ASCII
-	RU8PC855	IBM-PC Code Page 855 8-bit Latin/Cyrillic	SB, ASCII
-	RU8PC866	IBM-PC Code Page 866 8-bit Latin/Cyrillic	SB, ASCII
-	SE8ISO8859P3	ISO 8859-3 South European	SB, ASCII
-	US7ASCII	ASCII 7-bit American	SB, ASCII
-	US8PC437	IBM-PC Code Page 437 8-bit American	SB, ASCII
-	WE8DEC	DEC 8-bit West European	SB, ASCII
-	WE8DG	DG 8-bit West European	SB, ASCII
-	WE8ISO8859P1	ISO 8859-1 West European	SB, ASCII
-	WE8MACROMAN8S	Mac Server 8-bit Extended Roman8 West European	SB, ASCII
-	WE8NCR4970	NCR 4970 8-bit West European	SB, ASCII
-	WE8NEXTSTEP	NeXTSTEP PostScript 8-bit West European	SB, ASCII
-	WE8PC850	IBM-PC Code Page 850 8-bit West European	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
-	WE8ROMAN8	HP Roman8 8-bit West European	SB, ASCII
Universal	-	-	-
-	UTF8	Unicode 3.0 UTF-8 Universal character set, CESU-8 compliant	MB, ASCII, EURO

Table A–7 lists the other EBCDIC-based Oracle Database character sets. The list is ordered alphabetically within their language groups.

Table A–7 Other EBCDIC-based Database Character Sets

	Name	Description	Comments
Asian	-	-	-
-	TH8TISEBCDIC	Thai Industrial Standard 620-2533 - EBCDIC 8-bit	SB
-	ZHS16DBCS	IBM EBCDIC 16-bit Simplified Chinese	MB, UDC
-	ZHT16DBCS	IBM EBCDIC 16-bit Traditional Chinese	MB, UDC
Middle Eastern	-	-	-
-	AR8EBCDICX	EBCDIC XBASIC Server 8-bit Latin/Arabic	SB
-	IW8EBCDIC424	EBCDIC Code Page 424 8-bit Latin/Hebrew	SB
-	IW8EBCDIC1086	EBCDIC Code Page 1086 8-bit Hebrew	SB
-	TR8EBCDIC1026	EBCDIC Code Page 1026 8-bit Turkish	SB
-	WE8EBCDIC37C	EBCDIC Code Page 37 8-bit Oracle/c	SB
European	-	-	-
-	BLT8EBCDIC1112	EBCDIC Code Page 1112 8-bit Server Baltic Multilingual	SB
-	CL8EBCDIC1025	EBCDIC Code Page 1025 8-bit Cyrillic	SB
-	CL8EBCDIC1025C	EBCDIC Code Page 1025 Client 8-bit Cyrillic	SB
-	CL8EBCDIC1025S	EBCDIC Code Page 1025 Server 8-bit Cyrillic	SB
-	CL8EBCDIC1025X	EBCDIC Code Page 1025 (Modified) 8-bit Cyrillic	SB
-	CL8EBCDIC1158	EBCDIC Code Page 1158 8-bit Cyrillic	SB
-	D8BS2000	Siemens 9750-62 EBCDIC 8-bit German	SB
-	D8EBCDIC273	EBCDIC Code Page 273/1 8-bit Austrian German	SB
-	DK7SIEMENS9780X	Siemens 97801/97808 7-bit Danish	SB
-	DK8BS2000	Siemens 9750-62 EBCDIC 8-bit Danish	SB
-	DK8EBCDIC277	EBCDIC Code Page 277/1 8-bit Danish	SB
-	E8BS2000	Siemens 9750-62 EBCDIC 8-bit Spanish	SB
-	EE8EBCDIC870	EBCDIC Code Page 870 8-bit East European	SB
-	EE8EBCDIC870C	EBCDIC Code Page 870 Client 8-bit East European	SB
-	EL8EBCDIC875	EBCDIC Code Page 875 8-bit Greek	SB
-	EL8GCOS7	Bull EBCDIC GCOS7 8-bit Greek	SB
-	F8BS2000	Siemens 9750-62 EBCDIC 8-bit French	SB
-	F8EBCDIC297	EBCDIC Code Page 297 8-bit French	SB
-	I8EBCDIC280	EBCDIC Code Page 280/1 8-bit Italian	SB
-	S8BS2000	Siemens 9750-62 EBCDIC 8-bit Swedish	SB
-	S8EBCDIC278	EBCDIC Code Page 278/1 8-bit Swedish	SB
-	US8ICL	ICL EBCDIC 8-bit American	SB
-	US8BS2000	Siemens 9750-62 EBCDIC 8-bit American	SB
-	WE8EBCDIC924	Latin 9 EBCDIC 924	SB, EBCDIC
-	WE8EBCDIC37	EBCDIC Code Page 37 8-bit West European	SB

Table A–7 (Cont.) Other EBCDIC-based Database Character Sets

	Name	Description	Comments
-	WE8EBCDIC284	EBCDIC Code Page 284 8-bit Latin American/Spanish	SB
-	WE8EBCDIC285	EBCDIC Code Page 285 8-bit West European	SB
-	WE8EBCDIC1047	EBCDIC Code Page 1047 8-bit West European	SB
-	WE8EBCDIC1140C	EBCDIC Code Page 1140 8-bit West European	SB, EURO
-	WE8EBCDIC1148C	EBCDIC Code Page 1148 Client 8-bit West European	SB, EURO
-	WE8EBCDIC500C	EBCDIC Code Page 500 8-bit Oracle/c	SB
-	WE8EBCDIC500	EBCDIC Code Page 500 8-bit West European	SB
-	WE8EBCDIC871	EBCDIC Code Page 871 8-bit Icelandic	SB
-	WE8ICL	ICL EBCDIC 8-bit West European	SB
-	WE8GCOS7	Bull EBCDIC GCOS7 8-bit West European	SB
Universal	-	-	-
-	UTFE	EBCDIC form of Unicode 3.0 UTF-8 Universal character set (UTF-EBCDIC)	MB, EURO

Character Sets that Support the Euro Symbol

Table A–8 lists the character sets that support the Euro symbol.

Table A–8 Character Sets that Support the Euro Symbol

Character Set Name	Hexadecimal Code Value of the Euro Symbol
AL16UTF16	20AC
AL32UTF8	E282AC
AR8MSWIN1256	80
BLT8MSWIN1257	80
CL8EBCDIC1158	E1
CL8EBCDIC1158R	9F
CL8MSWIN1251	88
D8EBCDIC1141	9F
DK8EBCDIC1142	5A
EE8MSWIN1250	80
EL8EBCDIC423R	FD
EL8EBCDIC875R	DF
EL8ISO8859P7	A4
EL8MSWIN1253	80
F8EBCDIC1147	9F
I8EBCDIC1144	9F
IW8MSWIN1255	80
KO16KSC5601	A2E6
KO16KSCCS	D9E6

Table A–8 (Cont.) Character Sets that Support the Euro Symbol

Character Set Name	Hexadecimal Code Value of the Euro Symbol
KO16MSWIN949	A2E6
SE8EBCDIC1143	5A
TH8TISASCII	80
TR8MSWIN1254	80
UTF8	E282AC
UTFE	CA4653
VN8MSWIN1258	80
WE8BS2000E	9F
WE8EBCDIC1047E	9F
WE8EBCDIC1140	9F
WE8EBCDIC1140C	9F
WE8EBCDIC1145	9F
WE8EBCDIC1146	9F
WE8EBCDIC1148	9F
WE8EBCDIC1148C	9F
WE8EBCDIC924	9F
WE8ISO8859P15	A4
WE8MACROMAN8	DB
WE8MACROMAN8S	DB
WE8MSWIN1252	80
WE8PC858	DF
ZHS32GB18030	A2E3
ZHT16HKSCS	A3E1
ZHT16HKSCS31	A3E1
ZHT16MSWIN950	A3E1

Client-Only Character Sets

Table A–9 lists the Oracle Database character sets that are supported as client-only character sets. The list is ordered alphabetically within their respective language groups.

Table A–9 Client-Only Character Sets

	Name	Description	Comments
Asian	-	-	-
-	JA16EUCYEN	EUC 24-bit Japanese with '\' mapped to the Japanese yen character	MB
-	JA16MACSJIS	Mac client Shift-JIS 16-bit Japanese	MB
-	JA16SJISYEN	Shift-JIS 16-bit Japanese with '\' mapped to the Japanese yen character	MB, UDC

Table A–9 (Cont.) Client-Only Character Sets

	Name	Description	Comments
-	TH8MACTHAI	Mac Client 8-bit Latin/Thai	SB
-	ZHS16MACCGB231280	Mac client CGB2312-80 16-bit Simplified Chinese	MB
European	-	-	-
-	CH7DEC	DEC VT100 7-bit Swiss (German/French)	SB
-	CL8MACCYRILLIC	Mac Client 8-bit Latin/Cyrillic	SB
-	D7SIEMENS9780X	Siemens 97801/97808 7-bit German	SB
-	D7DEC	DEC VT100 7-bit German	SB
-	EEC8EUROASCI	EEC Targon 35 ASCI West European/Greek	SB
-	EEC8EUROPA3	EEC EUROPA3 8-bit West European/Greek	SB
-	EE8MACCROATIAN	Mac Client 8-bit Croatian	SB
-	EE8MACCE	Mac Client 8-bit Central European	SB
-	EL8PC737	IBM-PC Code Page 737 8-bit Greek/Latin	SB
-	EL8MACGREEK	Mac Client 8-bit Greek	SB
-	E7DEC	DEC VT100 7-bit Spanish	SB
-	E7SIEMENS9780X	Siemens 97801/97808 7-bit Spanish	SB
-	F7DEC	DEC VT100 7-bit French	SB
-	F7SIEMENS9780X	Siemens 97801/97808 7-bit French	SB
-	I7DEC	DEC VT100 7-bit Italian	SB
-	I7SIEMENS9780X	Siemens 97801/97808 7-bit Italian	SB
-	IS8MACICELANDICS	Mac Server 8-bit Icelandic	SB
-	IS8MACICELANDIC	Mac Client 8-bit Icelandic	SB
-	NL7DEC	DEC VT100 7-bit Dutch	SB
-	NDK7DEC	DEC VT100 7-bit Norwegian/Danish	SB
-	N7SIEMENS9780X	Siemens 97801/97808 7-bit Norwegian	SB
-	SF7DEC	DEC VT100 7-bit Finnish	SB
-	S7SIEMENS9780X	Siemens 97801/97808 7-bit Swedish	SB
-	S7DEC	DEC VT100 7-bit Swedish	SB
-	SF7ASCII	ASCII 7-bit Finnish	SB
-	WE8ISOICLUK	ICL special version ISO8859-1	SB
-	WE8MACROMAN8	Mac Client 8-bit Extended Roman8 West European	SB
-	WE8HP	HP LaserJet 8-bit West European	SB
-	YUG7ASCII	ASCII 7-bit Yugoslavian	SB
Middle Eastern	-	-	-
-	AR8ARABICMAC	Mac Client 8-bit Latin/Arabic	SB
-	AR8MUSSAD768	Mussa'd Alarabi/2 768 Server 8-bit Latin/Arabic	SB, ASCII
-	IW7IS960	Israeli Standard 960 7-bit Latin/Hebrew	SB

Table A–9 (Cont.) Client-Only Character Sets

	Name	Description	Comments
-	IW8MACHEBREW	Mac Client 8-bit Hebrew	SB
-	TR7DEC	DEC VT100 7-bit Turkish	SB
-	TR8MACTURKISH	Mac Client 8-bit Turkish	SB

Universal Character Sets

Table A–10 lists the Oracle Database 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–10 Universal Character Sets

Name	Description	Comments
AL16UTF16	Unicode 6.2 UTF-16 Universal character set	MB, EURO, FIXED
AL32UTF8	Unicode 6.2 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 (UTF-EBCDIC)	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 6, "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
ZHS32GB18030

You can use these character sets as the `source_char_set` or `dest_char_set` in the `CONVERT` function.

See *Oracle Database SQL Language Reference* for more information about the `CONVERT` function and "The CONVERT Function" on page 9-4.

Binary Subset-Superset Pairs

Oracle Database does not maintain a list of all subset-superset pairs of its character sets but it does maintain a list of binary subset-superset pairs that it recognizes when checking compatibility of two character sets.

Table A–11 lists all binary subset-superset relationships recognized by Oracle Database.

Table A–11 Binary Subset-Superset Pairs

Subset	Superset
AR8ARABICMACT	AR8ARABICMAC
AR8ISO8859P6	AR8ASMO8X
BLT8CP921	BLT8ISO8859P13
BLT8CP921	LT8MSWIN921
D7DEC	D7SIEMENS9780X
D7SIEMENS9780X	D7DEC
DK7SIEMENS9780X	N7SIEMENS9780X
I7DEC	I7SIEMENS9780X
I7SIEMENS9780X	IW8EBCDIC424
IW8EBCDIC424	IW8EBCDIC1086
KO16KSC5601	KO16MSWIN949
LT8MSWIN921	BLT8ISO8859P13
LT8MSWIN921	BLT8CP921
N7SIEMENS9780X	DK7SIEMENS9780X
US7ASCII	See Table A–12, " Binary Supersets of US7ASCII".
UTF8	AL32UTF8
WE8DEC	TR8DEC
WE8DEC	WE8NCR4970
WE8ISO8859P1	WE8MSWIN1252
WE8ISO8859P9	TR8MSWIN1254
WE8NCR4970	TR8DEC
WE8NCR4970	WE8DEC
WE8PC850	WE8PC858

US7ASCII is a special case because so many other character sets are supersets of it. Table A–12 lists all binary supersets of US7ASCII recognized by Oracle Database.

Table A–12 Binary Supersets of US7ASCII

Supersets a-e	Supersets e-n	Supersets r-z
AL32UTF8	EE8MACCROATIANS	RU8BESTA
AR8ADOS710	EE8MSWIN1250	RU8PC855
AR8ADOS720	EE8PC852	RU8PC866

Table A–12 (Cont.) Binary Supersets of US7ASCII

Supersets a-e	Supersets e-n	Supersets r-z
AR8APTEC715	EL8DEC	SE8ISO8859P3
AR8ARABICMACS	EL8ISO8859P7	TH8MACTHAIS
AR8ASMO8X	EL8MACGREEKS	TH8TISASCII
AR8ISO8859P6	EL8MSWIN1253	TR8DEC
AR8MSWIN1256	EL8PC437S	TR8MACTURKISHS
AR8MUSSAD768	EL8PC851	TR8MSWIN1254
AR8NAFITHA711	EL8PC869	TR8PC857
AR8NAFITHA721	ET8MSWIN923	US8PC437
AR8SAKHR706	HU8ABMOD	UTF8
AR8SAKHR707	HU8CWI2	VN8MSWIN1258
AZ8ISO8859PE	IN8ISCII	VN8VN3
BG8MSWIN	IS8PC861	WE8DEC
BG8PC437S	IW8ISO8859P8	WE8DG
BLT8CP921	IW8MACHEBREWS	WE8ISO8859P1
BLT8ISO8859P13	IW8MSWIN1255	WE8ISO8859P15
BLT8MSWIN1257	IW8PC1507	WE8ISO8859P9
BLT8PC775	JA16EUC	WE8MACROMAN8S
BN8BSCII	JA16EUCTILDE	WE8MSWIN1252
CDN8PC863	JA16SJIS	WE8NCR4970
CEL8ISO8859P14	JA16SJISTILDE	WE8NEXTSTEP
CL8ISO8859P5	JA16VMS	WE8PC850
CL8KOI8R	KO16KSC5601	WE8PC858
CL8KOI8U	KO16KSCCS	WE8PC860
CL8ISOIR111	KO16MSWIN949	WE8ROMAN8
CL8MACCYRILLICS	LA8ISO6937	ZHS16CGB231280
CL8MSWIN1251	LA8PASSPORT	ZHS16GBK
EE8ISO8859P2	LT8MSWIN921	ZHT16BIG5
EE8MACCES	LT8PC772	ZHT16CCDC
-	LT8PC774	ZHT16DBT
-	LV8PC1117	ZHT16HKSCS
-	LV8PC8LR	ZHT16MSWIN950
-	LV8RST104090	ZHT32EUC
-	N8PC865	ZHT32SOPS
-	NE8ISO8859P10	ZHT32TRIS
-	NEE8ISO8859P4	ZHS32GB18030

See Also: "Subsets and Supersets" on page 2-8 for discussion of what subsets and supersets of a character set are

Language and Character Set Detection Support

Table A–13 displays the languages and character sets that are supported by the Language and Character Set Detection utility (`LCSSCAN`) and the Globalization Development Kit (GDK).

Each language has several character sets that can be detected.

When the binary values for a language match two or more encodings that have a subset/superset relationship, the subset character set is returned. For example, if the language is German and all characters are 7-bit, then US7ASCII is returned instead of WE8MSWIN1252, WE8ISO8859P15, or WE8ISO8859P1.

When the character set is determined to be UTF-8, the Oracle Database character set UTF8 is returned by default unless 4-byte characters (supplementary characters) are detected within the text. If 4-byte characters are detected, then the character set is reported as AL32UTF8.

Table A–13 Languages and Character Sets Supported by LCSSCAN, and GDK

Language	Character Sets
Arabic	AL16UTF16, AL32UTF8, AR8ISO8859P6, AR8MSWIN1256, UTF8
Bulgarian	AL16UTF16, AL32UTF8, CL8ISO8859P5, CL8MSWIN1251, UTF8
Catalan	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Croatian	AL16UTF16, AL32UTF8, EE8ISO8859P2, EE8MSWIN1250, UTF8
Czech	AL16UTF16, AL32UTF8, EE8ISO8859P2, EE8MSWIN1250, UTF8
Danish	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Dutch	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
English	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Estonian	AL16UTF16, AL32UTF8, NEE8IOS8859P4, UTF8
Finnish	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
French	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
German	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Greek	AL16UTF16, AL32UTF8, EL8ISO8859P7, EL8MSWIN1253, UTF8
Hebrew	AL16UTF16, AL32UTF8, IW8ISO8859P8, IW8MSWIN1255, UTF8
Hindi	AL16UTF16, AL32UTF8, IN8ISCII, UTF8
Hungarian	AL16UTF16, AL32UTF8, EE8ISO8859P2, EE8MSWIN1250, UTF8
Indonesian	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Italian	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252

Table A–13 (Cont.) Languages and Character Sets Supported by LCSSCAN, and GDK

Language	Character Sets
Japanese	AL16UTF16, AL32UTF8, ISO2022-JP, JA16EUC, JA16SJIS, UTF8
Korean	AL16UTF16, AL32UTF8, ISO2022-KR, KO16KSC5601, KO16MSWIN949, UTF8
Latvian	AL16UTF16, AL32UTF8, NEE8ISO8859P4, UTF8
Lithuanian	AL16UTF16, AL32UTF8, NEE8ISO8859P4, UTF8
Malay	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Norwegian	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Persian	AL16UTF16, AL32UTF8, AR8MSWIN1256, UTF8
Polish	AL16UTF16, AL32UTF8, EE8ISO8859P2, EE8MSWIN1250, UTF8
Portuguese	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Romanian	AL16UTF16, AL32UTF8, EE8ISO8859P2, EE8MSWIN1250, UTF8
Russian	AL16UTF16, AL32UTF8, CL8ISO8859P5, CL8KOI8R, CL8MSWIN1251, RU8PC866, UTF8
Serbian	AL16UTF16, AL32UTF8, CL8ISO8859P5, CL8MSWIN1251, UTF8
Simplified Chinese	AL16UTF16, AL32UTF8, HZ-GB-2312, UTF8, ZHS16GBK, ZHS16CGB231280
Slovak	AL16UTF16, AL32UTF8, EE8ISO8859P2, EE8MSWIN1250, UTF8
Slovenian	AL16UTF16, AL32UTF8, EE8ISO8859P2, EE8MSWIN1250, UTF8
Spanish	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Swedish	AL16UTF16, AL32UTF8, US7ASCII, UTF8, WE8ISO8859P1, WE8ISO8859P15, WE8MSWIN1252
Thai	AL16UTF16, AL32UTF8, TH8TISASCII, UTF8
Traditional Chinese	AL16UTF16, AL32UTF8, UTF8, ZHT16MSWIN950
Turkish	AL16UTF16, AL32UTF8, TR8MSWIN1254, UTF8, WE8ISO8859P9
Ukranian	AL16UTF16, AL32UTF8, CL8ISO8859P5, CL8MSWIN1251, UTF8
Vietnamese	AL16UTF16, AL32UTF8, VN8VN3, UTF8

Linguistic Sorts

Oracle Database 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 are sorted differently from their ASCII values. For example, `ch` and `ll` are treated as a single character in `XSPANISH`.

All of the linguistic sorts can be also be performed as case-insensitive or accent-insensitive by appending `_CI` or `_AI` to the linguistic sort name.

Table A–14 lists the monolingual linguistic sorts supported by Oracle Database.

> **See Also:** Table A–1, " Oracle Database Supported Languages" on page A-2 for a list of the default sort for each language

Table A–14 Monolingual Linguistic Sorts

Basic Name	Extended Name	Special Cases
ARABIC	-	-
ARABIC_MATCH	-	-
ARABIC_ABJ_SORT	-	-
ARABIC_ABJ_MATCH	-	-
ASCII7	-	-
AZERBAIJANI	XAZERBAIJANI	i, I, lowercase i without dot, uppercase I with dot
BENGALI	-	-
BIG5	-	-
BINARY	-	-
BULGARIAN	-	-
CATALAN	XCATALAN	æ, AE, ß
CROATIAN	XCROATIAN	D, L, N, d, l, n, ß
CZECH	XCZECH	ch, CH, Ch, ß
CZECH_PUNCTUATION	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	-	-
LATIN	-	-
LATVIAN	-	-
LITHUANIAN	-	-
MALAY	-	-

Table A–14 (Cont.) Monolingual Linguistic Sorts

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
TURKISH	XTURKISH	æ, AE, ß
UKRAINIAN	-	-
UNICODE_BINARY	-	-
VIETNAMESE	-	-
WEST_EUROPEAN	XWEST_EUROPEAN	ß

Table A–15 lists the multilingual linguistic sorts available in Oracle Database. 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 sorts Korean and Latin-based characters, but it does not collate Chinese, Thai, or Japanese characters.

Table A–15 Multilingual LInguistic Sorts

Sort Name	Description
CANADIAN_M	Canadian French sort supports reverse secondary, special expanding characters
DANISH_M	Danish sort supports sorting uppercase characters before lowercase characters
FRENCH_M	French sort supports reverse sort for secondary
GENERIC_M	Generic sorting order which is based on ISO14651 and Unicode canonical equivalence rules but excluding compatible equivalence rules
JAPANESE_M	Japanese sort supports SJIS character set order and EUC characters which are not included in SJIS
KOREAN_M	Korean sort: Hangul characters are based on Unicode binary order. Hanja characters based on pronunciation order. All Hangul characters are before Hanja characters
SPANISH_M	Traditional Spanish sort supports special contracting characters
THAI_M	Thai sort supports swap characters for some vowels and consonants
SCHINESE_RADICAL_M	Simplified Chinese sort based on radical as primary order and number of strokes order as secondary order
SCHINESE_STROKE_M	Simplified Chinese sort uses number of strokes as primary order and radical as secondary order

Table A–15 (Cont.) Multilingual LInguistic Sorts

Sort Name	Description
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.

See Also: Chapter 5, "Linguistic Sorting and Matching"

Table A–16 illustrates UCA collations.

Table A–16 UCA Collations

Collation Name	UCA Version	Language	Collation Type	Default Setting for Collation Parameters
UCA0620_DUCET	6.2.0	All	DUCET	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_ROOT	6.2.0	All	CLDR root	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_CFRENCH	6.2.0	Canadian French	standard	_S4_VS_BY_NY_EN_FN_HN_DN_MN
UCA0620_DANISH	6.2.0	Danish	standard	_S4_VS_BN_NY_EN_FU_HN_DN_MN
UCA0620_JAPANESE	6.2.0	Japanese	standard	_S4_VS_BN_NY_EN_FN_HY_DN_MN
UCA0620_KOREAN	6.2.0	Korean	standard	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_SPANISH	6.2.0	Spanish	standard	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_TSPANISH	6.2.0	Spanish	traditional	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_THAI	6.2.0	Thai	standard	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_SCHINESE	6.2.0	Simplified Chinese	pinyin	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_SCHINESE1	6.2.0	Simplified Chinese	radical	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_SCHINESE2	6.2.0	Simplified Chinese	stroke	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_TCHINESE	6.2.0	Traditional Chinese	stroke	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0620_TCHINESE1	6.2.0	Traditional Chinese	radical	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_DUCET	6.1.0	All	DUCET	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_ROOT	6.1.0	All	CLDR root	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_CFRENCH	6.1.0	Canadian French	standard	_S4_VS_BY_NY_EN_FN_HN_DN_MN
UCA0610_DANISH	6.1.0	Danish	standard	_S4_VS_BN_NY_EN_FU_HN_DN_MN
UCA0610_JAPANESE	6.1.0	Japanese	standard	_S4_VS_BN_NY_EN_FN_HY_DN_MN
UCA0610_KOREAN	6.1.0	Korean	standard	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_SPANISH	6.1.0	Spanish	standard	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_TSPANISH	6.1.0	Spanish	traditional	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_THAI	6.1.0	Thai	standard	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_SCHINESE	6.1.0	Simplified Chinese	pinyin	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_SCHINESE1	6.1.0	Simplified Chinese	radical	_S4_VS_BN_NY_EN_FN_HN_DN_MN

Table A–16 (Cont.) UCA Collations

Collation Name	UCA Version	Language	Collation Type	Default Setting for Collation Parameters
UCA0610_SCHINESE2	6.1.0	Simplified Chinese	stroke	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_TCHINESE	6.1.0	Traditional Chinese	stroke	_S4_VS_BN_NY_EN_FN_HN_DN_MN
UCA0610_TCHINESE1	6.1.0	Traditional Chinese	radical	_S4_VS_BN_NY_EN_FN_HN_DN_MN

Calendar Systems

By default, most territory definitions use the Gregorian calendar system. Table A–17 lists the other calendar systems supported by Oracle Database.

Table A–17 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	US7ASCII
Ethiopian	Month DD "ቀን" YYYY	AL32UTF8

The Arabic Hijrah and English Hijrah calendars implemented in the Oracle Database are a variant of the tabular Islamic calendar in which the leap years are the 2nd, 5th, 7th, 10th, 13th, 16th, 18th, 21st, 24th, 26th, and 29th in the 30-years cycle and in which the 1st of Muharram 1 AH corresponds to the 16th of July 622 AD. Users can apply deviation days to modify the calendar to suit their requirements, for example, by following an alternative set of leap years. See "Customizing Calendars with the NLS Calendar Utility" on page 12-36 for more details about defining deviation days. The only difference between Arabic Hijrah and English Hijrah calendars are month names, which are written, correspondingly, in Arabic and in English transliteration.

Figure A–1 shows how March 27, 1998 appears in Japanese Imperial.

Figure A–1 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
```

Time Zone Region Names

Table A–18 shows the time zone region names in the time zone files for version 11 that are supplied with the Oracle Database. See Chapter 4, "Datetime Data Types and Time Zone Support" for more information regarding time zone files.

You can see the time zone region names by issuing the following statement:

```
SELECT DISTINCT(TZNAME) FROM V$TIMEZONE_NAMES;
```

Table A–18 Time Zone Region Names

Time Zone Name	In the Smaller Time Zone File?	Time Zone Name	In the Smaller Time Zone File?
Africa/Abidjan	No	Asia/Qatar	No
Africa/Accra	No	Asia/Qyzylorda	No
Africa/Addis_Ababa	No	Asia/Rangoon	No
Africa/Algiers	No	Asia/Riyadh	Yes
Africa/Asmara	No	Asia/Saigon	No
Africa/Asmera	No	Asia/Sakhalin	No
Africa/Bamako	No	Asia/Samarkand	No
Africa/Bangui	No	Asia/Seoul	Yes
Africa/Banjul	No	Asia/Shanghai	Yes
Africa/Bissau	No	Asia/Singapore	Yes
Africa/Blantyre	No	Asia/Taipei	Yes

Table A–18 (Cont.) Time Zone Region Names

Time Zone Name	In the Smaller Time Zone File?	Time Zone Name	In the Smaller Time Zone File?
Africa/Brazzaville	No	Asia/Tashkent	No
Africa/Bujumbura	No	Asia/Tbilisi	No
Africa/Cairo	Yes	Asia/Tehran	Yes
Africa/Casablanca	No	Asia/Tel_Aviv	Yes
Africa/Ceuta	No	Asia/Thimbu	No
Africa/Conakry	No	Asia/Thimphu	No
Africa/Dakar	No	Asia/Tokyo	Yes
Africa/Dar_es_Salaam	No	Asia/Ujung_Pandang	No
Africa/Djibouti	No	Asia/Ulaanbaator	No
Africa/Doula	No	Asia/Ulan_Bator	No
Africa/El_Aaiun	No	Asia/Urumqi	No
Africa/Freetown	No	Asia/Vientiane	No
Africa/Gaborone	No	Asia/Vladivostok	No
Africa/Harare	No	Asia/Yakutsk	No
Africa/Johannesburg	No	Asia/Yetaterinburg	No
Africa/Kampala	No	Asia/Yerevan	No
Africa/Khartoum	No	Atlantic/Azores	No
Africa/Kigali	No	Atlantic/Bermuda	No
Africa/Kinshasa	No	Atlantic/Canary	No
Africa/Lagos	No	Atlantic/Cape_Verde	No
Africa/Libreville	No	Atlantic/Faeroe	No
Africa/Lome	No	Atlantic/Faroe	No
Africa/Luanda	No	Atlantic/Jan_Mayen	No
Africa/Lubumbashi	No	Atlantic/Madeira	No
Africa/Lusaka	No	Atlantic/Reykjavik	Yes
Africa/Malabo	No	Atlantic/South_Georgia	No
Africa/Maputo	No	Atlantic/St_Helena	No
Africa/Maseru	No	Atlantic/Stanley	No
Africa/Mbabane	No	Australia/ACT	Yes
Africa/Mogadishu	No	Australia/Adelaide	Yes
Africa/Monrovia	No	Australia/Brisbane	Yes
Africa/Nairobi	No	Australia/Broken_Hill	Yes
Africa/Ndjamena	No	Australia/Canberra	Yes
Africa/Niamey	No	Australia/Currie	No
Africa/Nouakchott	No	Australia/Darwin	Yes
Africa/Ouagadougou	No	Australia/Eucla	No

Table A–18 (Cont.) Time Zone Region Names

Time Zone Name	In the Smaller Time Zone File?	Time Zone Name	In the Smaller Time Zone File?
Africa/Porto-Novo	No	Australia/Hobart	Yes
Africa/Sao_Tome	No	Australia/LHI	Yes
Africa/Timbuktu	No	Australia/Lindeman	Yes
Africa/Tripoli	Yes	Australia/Lord_Howe	Yes
Africa/Tunis	No	Australia/Melbourne	Yes
Africa/Windhoek	No	Australia/NSW	Yes
America/Adak	Yes	Australia/North	Yes
America/Anchorage	Yes	Australia/Perth	Yes
America/Anguilla	No	Australia/Queensland	Yes
America/Antigua	No	Australia/South	Yes
America/Araguaina	No	Australia/Sydney	Yes
America/Argentina/Buenos_Aires	No	Australia/Tasmania	Yes
America/Argentina/Catamarca	No	Australia/Victoria	Yes
America/Argentina/ComodRivadavia	No	Australia/West	Yes
America/Argentina/Cordoba	No	Australia/Yancowinna	Yes
America/Argentina/Jujuy	No	Brazil/Acre	Yes
America/Argentina/La_Rioja	Yes	Brazil/DeNoronha	Yes
America/Argentina/Mendoza	No	Brazil/East	Yes
America/Argentina/Rio_Gallegos	Yes	Brazil/West	Yes
America/Argentina/Salta	No	CET	Yes
America/Argentina/San_Juan	Yes	CST	Yes
America/Argentina/San_Luis	No	CST6CDT	Yes
America/Argentina/Tucuman	Yes	Canada/Atlantic	Yes
America/Argentina/Ushuaia	Yes	Canada/Central	Yes
America/Aruba	No	Canada/East-Saskatchewan	Yes
America/Asuncion	No	Canada/Eastern	Yes
America/Atikokan	No	Canada/Mountain	Yes
America/Atka	Yes	Canada/Newfoundland	Yes
America/Bahia	No	Canada/Pacific	Yes
America/Barbados	No	Canada/Saskatchewan	Yes
America/Belem	No	Canada/Yukon	Yes
America/Belize	No	Chile/Continental	Yes
America/Blanc-Sablon	No	Chile/EasterIsland	Yes
America/Boa_Vista	No	Cuba	Yes
America/Bogota	No	EET	Yes
America/Boise	No	EST	Yes

Table A–18 (Cont.) Time Zone Region Names

Time Zone Name	In the Smaller Time Zone File?	Time Zone Name	In the Smaller Time Zone File?
America/Buenos_Aires	No	EST5EDT	Yes
America/Cambridge_Bay	No	Egypt	Yes
America/Campo_Grande	No	Eire	Yes
America/Cancun	No	Etc/GMT	Yes
America/Caracas	No	Etc/GMT+0	Yes
America/Catamarca	No	Etc/GMT+1	Yes
America/Cayenne	No	Etc/GMT+10	Yes
America/Cayman	No	Etc/GMT+11	Yes
America/Chicago	Yes	Etc/GMT+12	Yes
America/Chihuahua	No	Etc/GMT+2	Yes
America/Coral_Harbour	No	Etc/GMT+3	Yes
America/Cordoba	No	Etc/GMT+4	Yes
America/Costa_Rica	No	Etc/GMT+5	Yes
America/Cuiaba	No	Etc/GMT+6	Yes
America/Curacao	No	Etc/GMT+7	Yes
America/Danmarkshavn	No	Etc/GMT+8	Yes
America/Dawson	No	Etc/GMT+9	Yes
America/Dawson_Creek	No	Etc/GMT-0	Yes
America/Denver	Yes	Etc/GMT-1	Yes
America/Detroit	Yes	Etc/GMT-10	Yes
America/Dominica	No	Etc/GMT-11	Yes
America/Edmonton	Yes	Etc/GMT-12	Yes
America/Eirunepe	Yes	Etc/GMT-13	Yes
America/El_Salvador	No	Etc/GMT-14	Yes
America/Ensenada	Yes	Etc/GMT-2	Yes
America/Fort_Wayne	Yes	Etc/GMT-3	Yes
America/Fortaleza	No	Etc/GMT-4	Yes
America/Glace_Bay	No	Etc/GMT-5	Yes
America/Godthab	No	Etc/GMT-6	yes
America/Goose_Bay	No	Etc/GMT-7	Yes
America/Grand_Turk	No	Etc/GMT-8	Yes
America/Grenada	No	Etc/GMT-9	Yes
America/Guadeloupe	No	Etc/GMT0	Yes
America/Guatemala	No	Etc/Greenwich	Yes
America/Guayaquil	No	Europe/Amsterdam	No
America/Guyana	No	-	-

Table A–18 (Cont.) Time Zone Region Names

Time Zone Name	In the Smaller Time Zone File?	Time Zone Name	In the Smaller Time Zone File?
America/Halifax	Yes	Europe/Andorra	No
America/Havana	Yes	Europe/Athens	No
America/Hermosillo	No	Europe/Belfast	Yes
America/Indiana/Indianapolis	Yes	Europe/Belgrade	No
America/Indiana/Knox	No	Europe/Berlin	No
America/Indiana/Marengo	No	Europe/Bratislava	No
America/Indiana/Petersburg	No	Europe/Brussels	No
America/Indiana/Tell_City	No	Europe/Bucharest	No
America/Indiana/Vevay	No	Europe/Budapest	No
America/Indiana/Vincennes	No	Europe/Chisinau	No
America/Indiana/Winamac	No	Europe/Copenhagen	No
America/Indianapolis	Yes	Europe/Dublin	Yes
America/Inuvik	No	Europe/Gibraltar	No
America/Iqaluit	No	Europe/Guernsey	Yes
America/Jamaica	Yes	Europe/Helsinki	No
America/Jujuy	No	Europe/Isle_of_Man	Yes
America/Juneau	No	Europe/Istanbul	Yes
America/Kentucky/Louisville	No	Europe/Jersey	Yes
America/Kentucky/Monticello	No	Europe/Kaliningrad	No
America/Knox_IN	No	Europe/Kiev	No
America/La_Paz	No	Europe/Lisbon	Yes
America/Lima	No	Europe/Ljubljana	No
America/Los_Angeles	Yes	Europe/London	Yes
America/Louisville	No	Europe/Luxembourg	No
America/Maceio	No	Europe/Madrid	No
America/Managua	No	Europe/Malta	No
America/Manaus	Yes	Europe/Mariehamn	No
America/Marigot	No	Europe/Minsk	No
America/Martinique	No	Europe/Monaco	No
America/Mazatlan	Yes	Europe/Moscow	Yes
America/Mendoza	No	Europe/Nicosia	No
America/Menominee	No	Europe/Oslo	No
America/Merida	No	Europe/Paris	No
America/Mexico_City	Yes	Europe/Podgorica	No
America/Miquelon	No	Europe/Prague	No
America/Moncton	No	Europe/Riga	No

Table A–18 (Cont.) Time Zone Region Names

Time Zone Name	In the Smaller Time Zone File?	Time Zone Name	In the Smaller Time Zone File?
America/Monterrey	Yes	Europe/Rome	No
America/Montevideo	No	Europe/Samara	No
America/Montreal	Yes	Europe/San_Marino	No
America/Montserrat	No	Europe/Sarajevo	No
America/Nassau	No	Europe/Simferopol	No
America/New_York	Yes	Europe/Skopje	No
America/Nipigon	No	Europe/Sofia	No
America/Nome	No	Europe/Stockholm	No
America/Noronha	Yes	Europe/Tallinn	No
America/North_Dakota/Center	No	Europe/Tirane	No
America/North_Dakota/New_Salem	No	Europe/Tiraspol	No
America/Panama	No	Europe/Uzhgorod	No
America/Pangnirtung	No	Europe/Vaduz	No
America/Paramaribo	No	Europe/Vatican	No
America/Phoenix	Yes	Europe/Vienna	No
America/Port-au-Prince	No	Europe/Vilnius	No
America/Port_of_Spain	No	Europe/Volgograd	No
America/Porto_Acre	No	Europe/Warsaw	Yes
America/Porto_Velho	No	Europe/Zagreb	No
America/Port_of_Spain	No	Europe/Zaporozhye	No
America/Porto_Acre	No	Europe/Zurich	No
America/Porto_Velho	No	GB	Yes
America/Puerto_Rico	No	GB-Eire	Yes
America/Rainy_River	No	GMT	Yes
America/Rankin_Inlet	No	GMT+0	Yes
America/Recife	No	GMT-0	Yes
America/Regina	Yes	GMT0	Yes
America/Resolute	No	Greenwich	Yes
America/Rio_Branco	Yes	HST	Yes
America/Rosario	No	Hongkong	Yes
America/Santiago	Yes	Iceland	Yes
America/Santo_Domingo	No	Indian/Antananarivo	No
America/Sao_Paulo	Yes	Indian/Chagos	No
America/Scoresbysund	No	Indian/Christmas	No
America/Shiprock	Yes	Indian/Cocos	No
America/St_Barthelemy	No	Indian/Comoro	No

Table A–18 (Cont.) Time Zone Region Names

Time Zone Name	In the Smaller Time Zone File?	Time Zone Name	In the Smaller Time Zone File?
America/St_Johns	Yes	Indian/Kerguelen	No
America/St_Kitts	No	Indian/Mahe	No
America/St_Lucia	No	Indian/Maldives	No
America/St_Thomas	No	Indian/Mauritius	No
America/St_Vincent	No	Indian/Mayotte	No
America/Swift_Current	No	Indian/Reunion	No
America/Tegucigalpa	No	Iran	Yes
America/Thule	No	Israel	Yes
America/Thunder_Bay	No	Jamaica	Yes
America/Tijuana	Yes	Japan	Yes
America/Tortola	No	Kwajalein	Yes
America/Vancouver	Yes	Libya	Yes
America/Virgin	No	MET	Yes
America/Whitehorse	Yes	MST	Yes
America/Winnipeg	Yes	MST7MDT	Yes
America/Yakutat	No	Mexico/BajaNorte	Yes
America/Yellowknife	No	Mexico/BajaSur	Yes
Antarctica/Casey	No	Mexico/General	Yes
Antarctica/Davis	No	NZ	Yes
Antarctica/DumontDUrville	No	NZ-CHAT	Yes
Antarctica/Mawson	No	Navajo	Yes
Antarctica/McMurdo	No	PRC	Yes
Antarctica/Palmer	No	PST	Yes
Antarctica/South_Pole	No	PST8PDT	Yes
Antarctica/Syowa	No	Pacific/Apia	No
Arctic/Longyearbyen	No	Pacific/Auckland	Yes
Asia/Aden	No	Pacific/Chatham	Yes
Asia/Almaty	No	Pacific/Easter	Yes
Asia/Amman	No	Pacific/Efate	No
Asia/Anadyr	No	Pacific/Enderbury	No
Asia/Aqtau	No	Pacific/Fakaofo	No
Asia/Aqtobe	No	Pacific/Fiji	No
Asia/Ashgabat	No	Pacific/Funafuji	No
Asia/Ashkhabad	No	Pacific/Galapagos	No
Asia/Baghdad	No	Pacific/Gambier	No
Asia/Bahrain	No	Pacific/Guadalcanal	No

Table A–18 (Cont.) Time Zone Region Names

Time Zone Name	In the Smaller Time Zone File?	Time Zone Name	In the Smaller Time Zone File?
Asia/Baku	No	Pacific/Guam	No
Asia/Bangkok	No	Pacific/Honolulu	Yes
Asia/Beirut	No	Pacific/Johnston	No
Asia/Bishkek	No	Pacific/Kiritimati	No
Asia/Brunei	No	Pacific/Kosrae	No
Asia/Calcutta	Yes	Pacific/Kwajalein	Yes
Asia/Choibalsan	No	Pacific/Majuro	No
Asia/Chongqing	No	Pacific/Marquesas	No
Asia/Chungking	No	Pacific/Midway	No
Asia/Colombo	No	Pacific/Nauru	No
Asia/Dacca	No	Pacific/Niue	No
Asia/Damascus	No	Pacific/Norfolk	No
Asia/Dhaka	No	Pacific/Noumea	No
Asia/Dili	No	Pacific/Pago_Pago	Yes
Asia/Dubai	No	Pacific/Palau	No
Asia/Dushanbe	No	Pacific/Pitcairn	No
Asia/Gaza	No	Pacific/Ponape	No
Asia/Harbin	No	Pacific/Rarotonga	No
Asia/Ho_Chi_Minh	No	Pacific/Rarotonga	No
Asia/Hong_Kong	Yes	Pacific/Saipan	No
Asia/Hovd	No	Pacific/Samoa	Yes
Asia/Irkutsk	No	Pacific/Tahiti	No
Asia/Istanbul	Yes	Pacific/Tarawa	No
Asia/Jakarta	No	Pacific/Tongatapu	No
Asia/Jayapura	No	Pacific/Truk	No
Asia/Jerusalem	Yes	Pacific/Wake	No
Asia/Kabul	No	Pacific/Wallis	No
Asia/Kamchatka	No	Pacific/Yap	No
Asia/Karachi	No	Poland	Yes
Asia/Kashgar	No	Portugal	Yes
Asia/Kathmandu	No	ROC	Yes
Asia/Katmandu	No	ROK	Yes
Asia/Kolkata	No	Singapore	Yes
Asia/Krasnoyarsk	No	Turkey	Yes
Asia/Kuala_Lumpur	No	US/Alaska	Yes
Asia/Kuching	No	US/Aleutian	Yes

Table A–18 (Cont.) Time Zone Region Names

Time Zone Name	In the Smaller Time Zone File?	Time Zone Name	In the Smaller Time Zone File?
Asia/Kuwait	No	US/Arizona	Yes
Asia/Macao	No	US/Central	Yes
Asia/Macau	No	US/East-Indiana	Yes
Asia/Magadan	No	US/Eastern	Yes
Asia/Makassar	No	US/Hawaii	Yes
Asia/Manila	No	US/Indiana-Starke	No
Asia/Muscat	No	US/Michigan	Yes
Asia/Nicosia	No	US/Mountain	Yes
Asia/Novosibirsk	No	US/Pacific	Yes
Asia/Omsk	No	US/Pacific-New	Yes
Asia/Oral	No	US/Samoa	Yes
Asia/Phnom_Penh	No	UTC	No
Asia/Pontianak	No	W-SU	Yes
Asia/Pyongyang	No	WET	Yes

See Also: "Choosing a Time Zone File" on page 4-15

Obsolete Locale Data

This section contains information about obsolete linguistic sorts, character sets, languages, and territories. The obsolete linguistic sort, language, and territory definitions are still available. However, they are supported for backward compatibility only; they may be desupported in a future release. You can obtain a listing of the obsolete character sets, languages, territories, and linguistic sorts for the current database release by querying the `V$NLS_VALID_VALUES` view.

Obsolete Linguistic Sorts

Table A–19 contains linguistic sorts that have been desupported as of Oracle Database 12*c*.

Table A–19 Obsolete Linguistic Sorts

Obsolete Sort Name	Replacement Sort
THAI_TELEPHONE	THAI_M
THAI_DICTIONARY	THAI_M
CANADIAN FRENCH	CANADIAN_M
JAPANESE	JAPANESE_M

Obsolete Territories

Table A–20 contains territories that have been desupported as of Oracle Database 12*c*.

Table A–20 Obsolete Territories

Obsolete Territory Name	Replacement Territory
CIS	RUSSIA
MACEDONIA	FYR MACEDONIA
YUGOSLAVIA	BOSNIA AND HERZEGOVINA, SERBIA, or MONTENEGRO
SERBIA AND MONTENEGRO	SERBIA or MONTENEGRO
CZECHOSLOVAKIA	CZECH REPUBLIC or SLOVAKIA

Obsolete Languages

Table A–21 contains languages that have been desupported in Oracle Database 12*c*.

Table A–21 Obsolete Languages

Obsolete Language Name	Replacement Language
BENGALI	BANGLA

Obsolete Character Sets and Replacement Character Sets

Table A–22 lists the obsolete character sets. If you reference any of these character sets in your code, then replace them with the new character set.

Table A–22 Obsolete Character Sets and Replacements

Old Character Set	Replacement Character Set
AL24UTFSS	UTF8, AL32UTF8
AR8ADOS710T	AR8ISO8859P6, AR8MSWIN1256, and AL32UTF8
AR8ADOS720T	AR8ISO8859P6, AR8MSWIN1256, and AL32UTF8
AR8APTEC715T	AR8ISO8859P6, AR8MSWIN1256, and AL32UTF8
AR8ASMO708PLUS	AR8ISO8859P6, AR8MSWIN1256, and AL32UTF8
AR8HPARABIC8T	AR8ISO8859P6, AR8MSWIN1256, and AL32UTF8
AR8MUSSAD768T	AR8ISO8859P6, AR8MSWIN1256, and AL32UTF8
AR8NAFITHA711T	AR8ISO8859P6, AR8MSWIN1256, and AL32UTF8
AR8NAFITHA721T	AR8ISO8859P6, AR8MSWIN1256, and AL32UTF8
AR8SAKHR707T	AR8ISO8859P6, AR8MSWIN1256, and AL32UTF8
AR8MSAWIN	AR8MSWIN1256
AR8XBASIC	AR8EBCDIC420S
CL8EBCDIC875S	CL8EBCDIC875R
CL8MSWINDOW31	CL8MSWIN1251
EL8EBCDIC875S	EL8EBCDIC875R
JVMS	JA16VMS
JEUC	JA16EUC

Table A–22 (Cont.) Obsolete Character Sets and Replacements

Old Character Set	Replacement Character Set
SJIS	JA16SJIS
JDBCS	JA16DBCS
KSC5601	KO16KSC5601
KDBCS	KO16DBCS
CGB2312-80	ZHS16CGB231280
CNS 11643-86	ZHT32EUC
JA16EUCFIXED	UTF8 and AL16UTF16. See also the following note in this section.
ZHS32EUCFIXED	UTF8 and AL16UTF16
ZHS16GBKFIXED	UTF8 and AL16UTF16
JA16DBCSFIXED	UTF8 and AL16UTF16
KO16DBCSFIXED	UTF8 and AL16UTF16
ZHS16DBCSFIXED	UTF8 and AL16UTF16
ZHS16CGB231280FIXED	UTF8 and AL16UTF16
ZHT16DBCSFIXED	UTF8 and AL16UTF16
KO16KSC5601FIXED	UTF8 and AL16UTF16
JA16SJISFIXED	UTF8 and AL16UTF16. See also the following note in this section.
ZHT16BIG5FIXED	UTF8 and AL16UTF16
ZHT32TRISFIXED	UTF8 and AL16UTF16

Note: The character sets `JA16EUCFIXED` (1830) and `JA16SJISFIXED` (1832) are supported on the database client side as follows:

- when specified in the `NLS_NCHAR` client environment variable,
- in the *ncharset* (last) parameter of the `OCIEnvNlsCreate()` call,
- or as a value of the `OCI_ATTR_CHARSET_ID` attribute of a bind or a define handle.

AL24UTFFSS Character Set Desupported

The Unicode Character Set AL24UTFFSS was desupported in Oracle9*i*. Oracle Database began offering the Unicode database character set UTF8 in Oracle8 and AL32UTF8 in Oracle9*i*. The AL32UTF8 character set has been updated to conform to Unicode 6.2 in Oracle Database 12*c* (12.1.0.2).

The migration path for an existing AL24UTFFSS database is to upgrade to UTF8 prior to upgrading to Oracle Database 9*i* or later. You can use the Character Set Scanner for data analysis in Oracle8 before attempting to migrate your existing database character set to UTF8.

Note: AL24UTFFSS was introduced in Oracle Database version 7 as the Unicode character set to support the UTF-8 encoding scheme, which was based on the Unicode standard 1.1, and which is now obsolete.

Updates to the Oracle Database Language and Territory Definition Files

Changes have been made to the content in some of the language and territory definition files since Oracle Database 10*g*. These updates are necessary to correct the legacy definitions that no longer meet the local conventions in some of the languages and territories that Oracle Database supports. These changes include modifications to the currency symbols, month names, and group separators. One example is the local currency symbol for Brazil. This was updated from `Cr$` to `R$` in Oracle Database 10*g*.

Please refer to the "Oracle Database Language and Territory Definition Changes" table documented in the `$ORACLE_HOME/nls/data/old/data_changes.html` file for a detailed list of the changes.

Oracle Database 12*c* customers should review their existing application code to make sure that the Oracle Database 12*c* locale definition files that provide the correct cultural conventions are being used. For customers who may not be able to make the necessary code changes to support their applications, Oracle Database continues to offer Oracle9*i* locale definition files with this release of Oracle Database.

To revert back to the Oracle9*i* language and territory behavior:

1. Shut down the database.
2. Run the script `cr9idata.pl` from the `$ORACLE_HOME/nls/data/old` directory.
3. Set the `ORA_NLS10` environment variable to the newly created `$ORACLE_HOME/nls/data/9idata` directory.
4. Restart the database.

Steps 2 and 3 will need to be repeated for all Oracle Database 12*c* clients that need to revert back to the Oracle9*i* definition files.

Oracle strongly recommends that customers use Oracle Database 12*c* locale definition files. Oracle9*i* locale definition files will be desupported in a future release.

B

Unicode Character Code Assignments

This appendix offers an introduction to Unicode character assignments. This appendix contains these topics:

- Unicode Code Ranges
- UTF-16 Encoding
- UTF-8 Encoding

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	-
Iindic, 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
Private 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	-	-
Indic, 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 nonapplicable 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. The AL16UTF16 character set in Oracle Database supports supplementary characters.

See Also: "Code Points and Supplementary Characters" on page 6-2

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

In Oracle Database, the AL32UTF8 character set supports 1-byte, 2-byte, 3-byte, and 4-byte values. In Oracle Database, the UTF8 character set supports 1-byte, 2-byte, and 3-byte values, but not 4-byte values.

Glossary

accent

A mark that changes the sound of a character. Because the common meaning of the word **accent** is associated with the stress or prominence of the character's sound, the preferred word in *Oracle Database Globalization Support Guide* is **diacritic**.

See also diacritic.

accent-insensitive linguistic sort

A linguistic sort that uses information only about base letters, not diacritics or case.

See also linguistic collation, base letter, diacritic, case.

AL16UTF16

The default Oracle Database character set for the SQL `NCHAR` data type, which is used for the national character set. It encodes Unicode data in the UTF-16BE (big endian) encoding scheme.

See also national character set, UTF-16.

AL32UTF8

An Oracle Database 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 scheme.

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 Database character set name is US7ASCII.

base letter

A character stripped of its diacritics and case. For example, the base letter for `a`, `A`, `ä`, and `Ä` is `a`.

See also diacritic.

binary collation

A type of collation that orders strings based on their binary representation (character encoding), treating each string as a simple sequences of bytes.

See also collation, linguistic collation, monolingual linguistic collation, multilingual linguistic collation, accent-insensitive linguistic sort, case-insensitive linguistic collation.

binary sorting

Ordering character strings using the binary collation.

byte semantics

Treatment of strings as a sequence of bytes. Offsets into strings and string lengths are expressed in bytes.

See also character semantics and length semantics.

canonical equivalence

A Unicode Standard term for describing that two characters or sequences of characters are to be semantically considered as the same character. Canonically equivalent characters cannot be distinguished when they are correctly rendered. For example, the precomposed character ñ (U+00F1 Latin Small Letter N With Tilde) is canonically equivalent to the sequence n (U+006E Latin Small Letter N) followed by ˜ (U+0303 Combining Tilde).

case

Refers to the condition of being uppercase or lowercase. For example, in a Latin alphabet, `A` is the uppercase form for `a`, which is the lowercase form.

case conversion

Changing a character from uppercase to lowercase or vice versa.

case-insensitive linguistic collation

A linguistic collation that uses information about base letters and diacritics but not case but not when determining the ordering of strings.

See also base letter, case, diacritic, linguistic collation.

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 monospaced A, proportional italic AA, cursive (longhand) 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. Each font will usually use a different shape, that is, a different glyph to represent the same character.

See also character code and glyph.

character classification

Information that provides details about the type of character associated with each character code. For example, a character can be uppercase, lowercase, punctuation, or control character.

character code

A character code is a sequence of bytes that represents a specific character. The sequence depends on the character 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 encoding form

A rule that assigns numbers to all characters in a character set.

character encoding scheme

A rule that maps numbers assigned by the character encoding form to particular sequences of bytes (character codes). For example, the UTF-16 encoding form has the big-endian encoding scheme (UTF-16BE) and the little-endian encoding scheme (UTF-16LE).

Most encoding forms have only one encoding scheme. Therefore, encoding form, encoding scheme, and encoding are often used interchangeably.

Oracle character sets correspond to character encoding schemes. For example, AL16UTF16 is the Oracle name for the UTF-16BE encoding scheme.

character repertoire

The characters that are available to be used, or encoded, in a specific character set.

character semantics

Treatment of strings as a sequence of characters. Offsets into strings and string lengths are expressed in characters (character codes).

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

A sequence 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).

client character set

The encoded character set used by the database client. A client character set can differ from the database character set. The database character set is sometimes called the **server 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.

code unit

The unit of encoded text for processing and interchange. The size of the code unit varies depending on the character encoding scheme. In most character encodings, a code unit is 1 byte. Important exceptions are UTF-16 and UCS-2, which use 2-byte code units, and wide character, which uses 4 bytes.

See also character encoding form.

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 collation, monolingual linguistic collation, multilingual linguistic collation, accent-insensitive linguistic sort, case-insensitive linguistic collation.

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.

Database Migration Assistant for Unicode (DMU)

An intuitive and user-friendly GUI tool to migrate your character set. It helps you streamline the migration process through an interface that minimizes the workload and ensures that all migration issues are addressed.

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 mainframe systems.

encoded character set

A character set with an associated character encoding scheme. An encoded character set specifies the byte sequence (character code) that is assigned to each character.

See also character encoding form.

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 (for example, translating error messages or user interface text from one language to another).

glyph

A glyph (font glyph) is a specific representation (shape) of a character. A character can have many different glyphs.

See also character.

ideograph

A symbol that represents an idea. Some writing systems use ideographs to represent words through their meaning instead of using letters to represent words through their sound. Chinese is an example of an ideographic writing system.

ISO

International Organization for Standardization. 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 collation standard that is designed for almost all languages of the world.

See also multilingual linguistic collation.

ISO/IEC 10646

A universal character set standard that defines the characters of most major scripts used in the modern world. ISO/IEC 10646 is kept synchronized with the Unicode Standard as far as character repertoire is concerned but it defines fewer properties and fewer text processing algorithms than the Unicode Standard.

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 Database 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 expressed as a number of characters (character codes) or as a number of bytes in the string.

See also character semantics and byte semantics.

linguistic collation

A type of collation that takes into consideration the standards and customs of spoken languages.

See also collation, linguistic sorting, monolingual linguistic collation, multilingual linguistic collation, accent-insensitive linguistic sort, case-insensitive linguistic collation.

linguistic index

An index built on a linguistic sort order.

linguistic sorting

Ordering character strings using a linguistic binary collation.

See also multilingual linguistic collation and monolingual linguistic collation.

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 collation

An Oracle Database collation that has two levels of comparison for strings. String are first ordered based on major values for their characters and if they are found equal in this comparison, they are further ordered based on minor values of their characters. Major values correspond roughly to base letters while minor values correspond to diacritics and case. Most European languages can be sorted with a monolingual collation, but monolingual collations are inadequate for Asian languages and for multilingual text.

See also multilingual linguistic collation.

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 UTF-8 Unicode encoding form. In UTF-8, 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 Database cannot tell whether 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 AL32UTF8.

See also single-byte character.

multibyte character string

A character string encoded in a multibyte character encoding scheme.

multibyte character encoding scheme

A character encoding scheme in which character codes may have more than one byte.

See also multibyte fixed-width character encoding scheme, multibyte varying-width character encoding scheme.

multibyte fixed-width character encoding scheme

A character encoding scheme in which each character code has the same fixed number of bytes, greater than one. AL16UTF16 is a multibyte fixed-width character set.

multibyte varying-width character encoding scheme

A character encoding scheme in which each character code has a number of bytes from a given range. The range is one to the maximum character width of the character set. Depending on the encoding scheme, the maximum character width of the character set may be 2, 3, or 4 bytes. For example, ZHT16BIG5 has character codes with one or two bytes. UTF8 has character codes with one, two, or three bytes. AL32UTF8 has character codes with one, two, three, or four bytes. Oracle does not support encoding schemes with more than 4 bytes per character code.

multilingual linguistic collation

An Oracle Database collation that evaluates strings on three levels. Asian languages require a multilingual linguistic collation even if data exists in only one language. Multilingual linguistic collations are also used when data exists in several languages.

In multilingual collations, strings are first ordered based on primary weights, then, if necessary, secondary weights, then tertiary weights. For letters, primary weights correspond to base letters, secondary weights to diacritics, and tertiary weights to case and specific decoration, such as circle around the character. For ideographic scripts weights may represent other character variations.

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 AL16UTF16 and UTF8 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 Oracle Database. You can create user-defined NLB files with Oracle Locale Builder.

See also Oracle Locale Builder and NLT files.

NLS

National Language Support. NLS enables users to interact with the database in their native languages. It also enables applications to run in different linguistic and cultural environments. The term has been replaced by the terms globalization and localization.

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.

replacement character

A character used during character conversion when the source character is not available in the target character set. For example, `?` (question mark) is often used as the default replacement character in Oracle character sets.

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, but the use of ISO 8859-1 restricts the multilingual support. Thai or Chinese could not be added to the group.

SQL CHAR data types

Includes `CHAR`, `VARCHAR`, `VARCHAR2`, `CLOB`, and `LONG` data types.

SQL NCHAR data types

Includes `NCHAR`, `NVARCHAR2`, and `NCLOB` data types.

script

A particular system of writing. 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. Examples 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 Database 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 string encoded in a single-byte character encoding scheme. The term may also be used to describe a multibyte varying-width character string that happens to consist only of single-byte character codes.

See also multibyte varying-width character encoding scheme.

sort

An ordering of strings. This can be based on requirements from a locale instead of the binary representation of the strings, which is called a linguistic sort, or based on binary coded values, which is called a binary sort.

See also multilingual linguistic collation and monolingual linguistic collation.

supplementary characters

The first version of the Unicode Standard was a 16-bit, fixed-width encoding that used two bytes to encode each character. This enabled 65,536 characters to be represented. However, more characters need to be supported because of the large number of Asian ideograms.

Unicode Standard version 3.1 defined supplementary characters to meet this need by extending the numbering range for characters from 0000-FFFF hexadecimal to 0000-10FFFF hexadecimal. Unicode 3.1 began using two 16-bit code units (also known as **surrogate pairs**) to represent a single supplementary character in the UTF-16 form. This enabled an additional 1,048,576 characters to be defined. The Unicode 3.1 standard added the first group of 44,944 supplementary characters. More were added with subsequent versions of the Unicode Standard.

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

An obsolete form for an ISO/IEC 10646 standard character set encoding form. Currently used to mean the UTF-16 encoding form without support for surrogate pairs.

UCS-4

An obsolete name for an ISO/IEC 10646 standard encoding form, synonymous with UTF-32.

Unicode Standard

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

Unicode Standard also defines various text processing algorithms and related character properties to aid in complex script processing of scripts such as Arabic or Devanagari (Hindi).

Unicode database

A database whose database character set is AL32UTF8 or UTF8.

Unicode code point

A value in the Unicode codespace, which ranges from 0 to 0x10FFFF. Unicode assigns a unique code point to every character.

Unicode data type

A SQL `NCHAR` data type (`NCHAR`, `NVARCHAR2`, and `NCLOB`). You can store Unicode characters in columns of these data types even if the database character set is not based on the Unicode Standard.

unrestricted multilingual support

The ability to use as many languages as desired. A universal character set, such as Unicode Standard, helps to provide unrestricted multilingual support because it supports a very large character repertoire, encompassing most modern languages of the world.

UTFE

An Oracle character set implementing a 4-byte subset of the Unicode UTF-EBCDIC encoding form, used only on EBCDIC platforms and deprecated.

UTF8

The UTF8 Oracle character set encodes characters in one, two, or three bytes. The UTF8 character set supports Unicode 3.0 and implements the CESU-8 encoding scheme. 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. UTF8 is deprecated.

UTF-8

The 8-bit encoding form and scheme of the Unicode Standard. It is a multibyte varying-width encoding. One Unicode character can be 1 byte, 2 bytes, 3 bytes, or 4 bytes in the 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. The Oracle Database character set that implements UTF-8 is AL32UTF8.

UTF-16

The 16-bit encoding form of Unicode. One Unicode character can be one or two 2-code units in the UTF-16 encoding. Characters (including ASCII characters) from European scripts and most Asian scripts are represented by one code unit (2 bytes). Supplementary characters are represented by two code units (4 bytes). The Oracle Database character sets that implement UTF-16 are AL16UTF16 and AL16UTF16LE. AL16UTF16 implements the big-endian encoding scheme of the UTF-16 encoding form (more significant byte of each code unit comes first in memory). AL16UTF16 is a valid national character set. AL16UTF16LE implements the little-endian UTF-16 encoding scheme. It is a conversion-only character set, valid only in character set conversion functions such as SQL `CONVERT` or PL/SQL `UTL_I18N.STRING_TO_RAW`.

Note that most SQL string processing functionality treats each UTF-16 code unit in AL16UTF16 as a separate character. The functions `INSTR4`, `SUBSTR4`, and `LENGTH4` are an exception.

wide character

A multibyte fixed-width character format that is useful for extensive text processing because it enables data to be processed in consistent, fixed-width chunks. Multibyte varying-width character values may be internally converted to the wide character format for faster processing.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X