

Informix Guide to GLS Functionality

Version 9.1
March 1997
Part No. 000-4818

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

Copyright © 1981-1997 by Informix Software, Inc. or their subsidiaries, provided that portions may be copyrighted by third parties, as set forth in documentation. All rights reserved.

The following are worldwide trademarks of Informix Software, Inc., or its subsidiaries, registered in the United States of America as indicated by “®,” and in numerous other countries worldwide:

INFORMIX®; INFORMIX®-OnLine Dynamic Server™; DataBlade®

The following are worldwide trademarks of the indicated owners or their subsidiaries, registered in the United States of America as indicated by “®,” and in numerous other countries worldwide:

Adobe Systems Incorporated: PostScript®

International Business Machines Corporation: IBM®

Microsoft Corporation: Microsoft®; MS®; MS-DOS®; CodeView®; MS Windows™; Windows™;

Windows NT™; ODBC™; Visual Basic™; Visual C++™

Microsoft Memory Management Product: HIMEM.SYS

(“DOS” as used herein refers to MS-DOS and/or PC-DOS operating systems.)

X/Open Company Ltd.: UNIX®; X/Open®

All other marks or symbols are registered trademarks or trademarks of their respective owners.

Documentation Team: Twila Booth, Diana Chase, Karin Kristenson, Dawn Maneval

To the extent that this software allows the user to store, display, and otherwise manipulate various forms of data, including, without limitation, multimedia content such as photographs, movies, music and other binary large objects (blobs), use of any single blob may potentially infringe upon numerous different third-party intellectual and/or proprietary rights. It is the user's responsibility to avoid infringements of any such third-party rights.

RESTRICTED RIGHTS/SPECIAL LICENSE RIGHTS

Software and documentation acquired with US Government funds are provided with rights as follows: (1) if for civilian agency use, with Restricted Rights as defined in FAR 52.227-19; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by negotiated vendor license as prescribed in DFAR 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce the legend.

Table of Contents

Introduction

About This Manual	3
Organization of This Manual	4
Types of Users	5
Software Dependencies	5
Documentation Conventions	6
Typographical Conventions	6
Icon Conventions	7
Command-Line Conventions	8
Sample-Code Conventions	11
Character-Representation Conventions	12
Additional Documentation	15
On-Line Manuals	15
Printed Manuals	16
Error Message Files	16
Documentation Notes, Release Notes, Machine Notes	17
Related Reading	17
Compliance with Industry Standards	18
Informix Welcomes Your Comments	18

Chapter 1

GLS Fundamentals

What Is the GLS Feature?	1-3
GLS Support by Informix Products	1-5
A GLS Locale	1-8
How Do You Set a GLS Locale?	1-18
The Default Locale	1-18
GLS Locale Names	1-20
Locales in the Client/Server Environment	1-22
A Nondefault Locale	1-28
How Do Informix Products Use GLS Locales?	1-30
Supporting Non-ASCII Characters	1-30
Establishing a Database Connection	1-31

Performing Code-Set Conversion	1-40
Locating Message Files.	1-49
How Do You Customize Client End-User Formats?	1-49
Customizing Date and Time End-User Formats	1-50
Customizing Monetary Values	1-53

Chapter 2 GLS Environment Variables

GLS-Related Environment Variables	2-4
CC8BITLEVEL	2-5
CLIENT_LOCALE	2-6
DBDATE	2-7
DBLANG	2-15
DB_LOCALE	2-17
DBMONEY.	2-19
DBTIME.	2-20
ESQLMF.	2-22
GLS8BITFSYS	2-23
GL_DATE	2-25
GL_DATETIME	2-35
SERVER_LOCALE	2-41

Chapter 3 SQL Features

Naming Database Objects	3-4
Rules for Identifier Names	3-4
Identifiers That Support Non-ASCII Characters	3-5
Valid Characters in Identifier Names	3-8
Non-ASCII Characters in Delimited Identifiers	3-10
Multibyte Characters and Identifier Length.	3-10
Using Character Data Types	3-11
The NCHAR Data Type	3-11
The NVARCHAR Data Type	3-13
Performance Considerations for NCHAR and NVARCHAR	3-16
Other Character Data Types	3-17
Collating Character Data	3-20
Collation Order in CREATE INDEX	3-20
Collation Order in SELECT Statements	3-21
Handling Character Data	3-33
Specifying Quoted Strings	3-33
Specifying Comments	3-34
Specifying Column Substrings	3-34
Specifying Arguments to the TRIM Function	3-40

Using SQL Length Functions	3-40
The LENGTH Function	3-40
The OCTET_LENGTH Function	3-43
The CHAR_LENGTH Function	3-45
Handling MONEY Columns	3-48
Default Values for the Scale Parameter	3-48
Format of Currency Notation	3-49
Using Data Manipulation Statements	3-49
Specifying Conditions in the WHERE Clause	3-50
Specifying Era-Based Dates.	3-50
Loading and Unloading Data	3-51

Chapter 4 **INFORMIX-Universal Server Features**

Using the Server Locale	4-3
Generating Non-ASCII Filenames	4-4
Performing Code-Set Conversion	4-6
Support for Locales by Universal Server Utilities	4-8
Locale Environment Variables.	4-9
Non-ASCII Characters in Command-Line Arguments	4-10
Universal Server Data Types	4-11
SQL-92 Data Types	4-12
LVARCHAR Data Type	4-13
Smart Large Objects	4-13
Complex Data Types	4-14
Opaque Data Types	4-14
Distinct Data Types	4-16
SQL Identifiers That Support Non-ASCII Characters	4-17
Introducing the Informix GLS API	4-18
Compliance with Industry Standards	4-18
How to Use the GLS API	4-19
Compiling and Linking the GLS API	4-19
How to Internationalize Programs with the GLS API	4-20

Chapter 5 **INFORMIX-OnLine Dynamic Server Features**

Using the Server Locale	5-3
Generating Non-ASCII Filenames	5-4
Performing Code-Set Conversion	5-6
Support for Locales by OnLine Utilities	5-8
Locale Environment Variables.	5-8
Non-ASCII Characters in Command-Line Arguments	5-9
Enhancements to the onmode Utility	5-10

Chapter 6	INFORMIX-SE Features	
	Using the Server Locale	6-3
	Generating Non-ASCII Filenames	6-4
	Performing Code-Set Conversion	6-6
	Naming a Database	6-7
	Support for Locales by SE Utilities	6-8
	Locale Environment Variables	6-8
	Non-ASCII Characters in Command-Line Arguments	6-9
 Chapter 7	 General SQL API Features	
	Non-ASCII Characters in ESQL Source Files	7-3
	Generating Non-ASCII Filenames	7-4
	Using Non-ASCII Characters in Source Code	7-6
	Enhanced ESQL Library Functions	7-7
	DATE-Format Functions	7-8
	DATETIME-Format Functions	7-15
	Numeric-Format Functions	7-17
	String Functions	7-23
	GLS-Specific Error Messages	7-23
	Establishing a Database Connection	7-24
	Sending Client-Locale Information	7-24
	Checking for Connection Warnings	7-24
	Avoiding Partial Characters	7-25
	Copying Character Data	7-26
	Using Code-Set Conversion	7-26
 Chapter 8	 INFORMIX-ESQL/C Features	
	Non-ASCII Characters in Host Variables	8-4
	Locale-Sensitive Character Data Types	8-5
	Extended ESQL/C Library Functions	8-7
	The esqlmf Filter	8-7
	Filtering Non-ASCII Characters	8-8
	Invoking the ESQL/C Filter	8-9
	Handling Code-Set Conversion	8-12
	Writing TEXT Values	8-13
	Using the DESCRIBE Statement	8-14
	Using the TRIM Function	8-16
	Using TRIM with INFORMIX-SE	8-17
	Using TRIM with INFORMIX-OnLine Dynamic Server	8-17
	Using TRIM with INFORMIX-Universal Server	8-18

Appendix A	Managing GLS Files
	Glossary
	Index

Introduction

About This Manual	3
Organization of This Manual	4
Types of Users	5
Software Dependencies	5
Documentation Conventions	6
Typographical Conventions	6
Icon Conventions	7
Comment Icons	7
Feature and Platform Icons	8
Command-Line Conventions	8
How to Read a Command-Line Diagram	10
Sample-Code Conventions	11
Character-Representation Conventions	12
Single-Byte Characters	12
Multibyte Characters	13
Single-Byte and Multibyte Characters in the Same String	13
White Space in Strings	14
Trailing White Spaces	14
Additional Documentation	15
On-Line Manuals	15
Printed Manuals	16
Error Message Files	16
Documentation Notes, Release Notes, Machine Notes	17
Related Reading	17
Compliance with Industry Standards	18
Informix Welcomes Your Comments	18

R

ead this introduction for an overview of the information provided in this manual and for an understanding of the documentation conventions used.

About This Manual

The *Guide to GLS Functionality* describes the Global Language Support (GLS) feature available in Informix products. The GLS feature allows Informix application-programming interfaces (APIs) and Informix database servers to handle different languages, cultural conventions, and code sets. This manual describes only the language-related topics that are unique to GLS.

The GLS feature provides support for the following language-related topics:

- Collation order of characters
- Definition of uppercase and lowercase conventions
- Non-ASCII characters as character data
- Culture-specific formatting for numeric, monetary, date, and time values

This manual is platform independent. It provides GLS information for both the UNIX and the Microsoft Windows environments.

Organization of This Manual

This manual includes the following chapters:

- This Introduction provides an overview of the manual and describes the documentation conventions used.
- [Chapter 1, “GLS Fundamentals,”](#) introduces GLS concepts and terms that are used throughout the manual, explains how to use and specify a locale for Informix products, and explains how to use code-set conversion to convert character data between a client and a database.
- [Chapter 2, “GLS Environment Variables,”](#) describes the GLS-related environment variables.
- [Chapter 3, “SQL Features,”](#) explains how the GLS environment affects the Informix implementation of SQL.
- [Chapter 4, “INFORMIX-Universal Server Features,”](#) explains how the GLS environment affects INFORMIX-Universal Server.
- [Chapter 5, “INFORMIX-OnLine Dynamic Server Features,”](#) explains how the GLS environment affects INFORMIX-OnLine Dynamic Server.
- [Chapter 6, “INFORMIX-SE Features,”](#) explains how the GLS environment affects the INFORMIX-SE database server.
- [Chapter 7, “General SQL API Features,”](#) explains how the GLS environment affects the Informix SQL application programming interfaces (APIs), INFORMIX-ESQL/C and INFORMIX-ESQL/COBOL.
- [Chapter 8, “INFORMIX-ESQL/C Features,”](#) explains features that are specific to INFORMIX-ESQL/C.
- [Appendix A, “Managing GLS Files,”](#) provides information to help you manage GLS files and includes information on how to use the **glfiles** utility.
- [“Glossary”](#) provides a listing of terms and definitions used in products that provide GLS.

Types of Users

This manual is written for application developers and system administrators who want to use the GLS environment with Informix products. Informix products use the U.S. 8859-1 English locale as the default locale. This manual is primarily intended for those users who need to use Informix products with a nondefault locale. It is assumed that you are familiar with the operating system and the Informix products that are being used.

Software Dependencies

This manual assumes that you are using one of the following Informix products:

- C-ISAM, Version 7.2x
- DB-Access, Version 7.2x or Version 9.x
- ESQL/C, Version 7.2x or Version 9.x
- ESQL/COBOL, Version 7.2x
- INFORMIX-OnLine Dynamic Server, Version 7.2x
- INFORMIX-SE, Version 7.2x
- INFORMIX-Universal Server, Version 9.x

In this manual, all instances of Universal Server refer to INFORMIX-Universal Server.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other Informix manuals.

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Command-line conventions
- Sample-code conventions
- Character-representation conventions

Typographical Conventions

This manual uses the following standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
<i>italics</i>	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
monospace	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
◆	This symbol indicates the end of feature-, product-, platform-, or compliance-specific information.




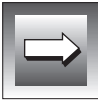

Tip: When you are instructed to “enter” characters or to “execute” a command, immediately press RETURN after the entry. When you are instructed to “type” the text or to “press” other keys, no RETURN is required.

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.





Comment Icons

Comment icons identify warnings, important notes, or tips. This information is always displayed in italics.

Icon	Description
	The <i>warning</i> icon identifies vital instructions, cautions, or critical information.
	The <i>important</i> icon identifies significant information about the feature or operation that is being described.
	The <i>tip</i> icon identifies additional details or shortcuts for the functionality that is being described.

Feature and Platform Icons

Feature and platform icons identify paragraphs that contain feature-specific or platform-specific information.

Icon	Description
	Identifies information that is specific to an Informix Asian Language Support (ALS) database or application.
	Identifies information that relates to the Native Language Support (NLS) feature.
	Identifies information that is specific to the UNIX platform.
	Identifies information that is specific to Windows environments: Windows NT, Windows 95, and Windows 3.1 (Win32s).

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the feature- or platform-specific information.


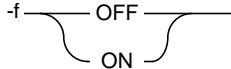
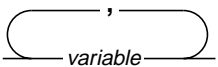
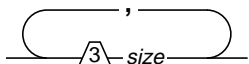
Command-Line Conventions

This section defines and illustrates the format of commands that are available in Informix products. These commands have their own conventions, which might include alternative forms of a command, required and optional parts of the command, and so forth.

Each diagram displays the sequences of required and optional elements that are valid in a command. A diagram begins at the upper-left corner with a command. It ends at the upper-right corner with a vertical line. Between these points, you can trace any path that does not stop or back up. Each path describes a valid form of the command. You must supply a value for words that are in *italics*.

You might encounter one or more of the following elements on a command-line path.

Element	Description
command	This required element is usually the product name or other short word that invokes the product or calls the compiler or preprocessor script for a compiled Informix product. It might appear alone or precede one or more options. You must spell a command exactly as shown and use lowercase letters.
<i>variable</i>	A word in italics represents a value that you must supply, such as a database, file, or program name. A table following the diagram explains the value.
-flag	A flag is usually an abbreviation for a function, menu, or option name or for a compiler or preprocessor argument. You must enter a flag exactly as shown, including the preceding hyphen.
.ext	A filename extension, such as .sql or .cob , might follow a variable that represents a filename. Type this extension exactly as shown, immediately after the name of the file. The extension might be optional in certain products.
(, ; + * - /)	Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.
<div>Privileges p. 5-17</div> <div>Privileges</div>	A reference in a box represents a subdiagram. Imagine that the subdiagram is spliced into the main diagram at this point. When a page number is not specified, the subdiagram appears on the same page.
<div>Standard DBDATE Formats see SQLR</div>	A reference to SQLR in this manual refers to the Informix Guide to SQL: Reference . Imagine that the subdiagram is spliced into the main diagram at this point.
— ALL —	A shaded option is the default action.
—▶▶—	Syntax within a pair of arrows indicates a subdiagram.

Element	Description
	The vertical line terminates the command.
	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items.
	A gate ($\sqrt{3}$) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. Here you can specify <i>size</i> no more than three times within this statement segment.

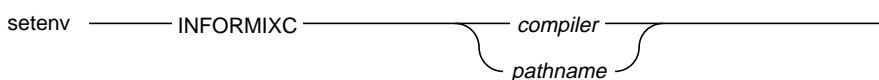
(2 of 2)

How to Read a Command-Line Diagram

Figure 1 shows a command-line diagram that uses some of the elements that are listed in the previous table.

Figure 1

Example of a Command-Line Diagram



To construct a command correctly, start at the top left with the command **setenv**. Then follow the diagram to the right, including the elements that you want. The elements in the diagram are case sensitive. ♦

These platforms use a different syntax to set environment variables. They do not use the **setenv** command. Many Informix products on Windows environments use the SETNET32 graphical tool to set environment variables. However, the following steps are still useful to show how to follow command-line syntax. ♦

Figure 1 on page -10 diagrams the following steps:

1. Type the word `setenv`.
2. Type the word `INFORMIXC`.
3. Supply either a compiler name or pathname.
After you choose *compiler* or *pathname*, you come to the terminator.
Your command is complete.
4. Press RETURN to execute the command.

Sample-Code Conventions

Examples of SQL code occur throughout this manual. Except where noted, the code is not specific to any single Informix application development tool. If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores7
:
:
DELETE FROM customer
      WHERE customer_num = 121
:
:
COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using the Query-language option of DB-Access, you must delimit multiple statements with semicolons. If you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement.



Tip: *Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.*

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the manual for your product.

Character-Representation Conventions

Throughout this manual, examples show how single-byte and multibyte characters appear. Because multibyte characters are usually ideographic (such as Japanese or Chinese characters), this manual does not use the actual multibyte characters. Instead, it uses ASCII characters to represent both single-byte and multibyte characters. This section provides general information about how this manual represents multibyte and single-byte characters abstractly. For a definition of multibyte and single-byte characters, see the [“Glossary.”](#)

Single-Byte Characters

This manual represents single-byte characters as a series of lowercase letters. The format for representing one single-byte character abstractly is:

a

In this format, a stands for any single-byte character, not for the letter “a” itself.

The format for representing a string of single-byte characters is as follows:

a . . . z

In this format, a stands for the first character in the string, and z stands for the last character in the string. For example, if the string `Ludwig` consists of single-byte characters, the following format represents this 6-character string abstractly:

abcdef

Tip: The letter “s” does not appear in alphabetical sequences that represent strings of single-byte characters. The manual reserves the letter “s” as a symbol that represents a single-byte white-space character. For further information, see [“White Space in Strings” on page 14.](#)



Multibyte Characters

This manual does not attempt to show the actual appearance of multibyte characters in text, examples, or diagrams. Instead, the following convention shows abstractly how multibyte characters are stored:

$$A^1 \dots A^n$$

One to four identical uppercase letters, each followed by a different superscript number, represent one multibyte character. The superscripts show the first to the n th byte of the multibyte character, where n has values between two and four. For example, the following symbols represent a multibyte character that consists of two bytes:

$$A^1 A^2$$

The following notation represents a multibyte character that consists of four bytes (the maximum length of a multibyte character):

$$A^1 A^2 A^3 A^4$$

The following example shows a string of multibyte characters in an SQL statement:

```
CREATE DATABASE A1A2B1B2C1C2D1D2E1E2;
```

This statement creates a database whose name consists of five multibyte characters, each of which is two bytes long. For more information on how to use multibyte characters in SQL identifiers, see [“Naming Database Objects” on page 3-4](#).

Single-Byte and Multibyte Characters in the Same String

If you are using a multibyte code set, a given string might be composed of both single-byte and multibyte characters. To represent these mixed strings, this manual simply combines the formats for multibyte and single-byte characters.

For example, suppose that you have a string with four characters. The first and fourth characters are single-byte characters while the second and third characters are multibyte characters that consist of two bytes each. The following format represents this string:

$$aA^1A^2B^1B^2b$$

White Space in Strings

White space is a series of one or more space characters. A GLS locale defines what characters are considered to be space characters. For example, both the TAB and blank might be defined as space characters in one locale, but certain combinations of the CTRL key and another character might be defined as space characters in a different locale.

The convention for representing single-byte white spaces in this manual is the letter “s.” The following notation represents one single-byte white space:

s

In the ASCII code set, an example of a single-byte white space is the blank character (ASCII code number 32). To represent a string that consists of two ASCII blank characters, the manual uses the following notation:

s s

The following notation represents a multibyte white-space character:

$s^1 \dots s^n$

In this format s^1 represents the first byte of the white-space character, while s^n represents the last byte of the white-space character, where n has values between two and four. For example, the following notation represents one 4-byte white-space character:

$s^1 s^2 s^3 s^4$

Trailing White Spaces

Combinations of characters and white spaces can occur in quoted strings, in CHAR columns that contain fewer characters than the defined length of the column, and in other situations. For example, if a CHAR(5) column in a single-byte code set contains a string of three characters, the string is extended with two white spaces so that its length is equal to the defined length of the column, as follows:

abcss

The following example shows the representation for a string of five characters (three characters of data and two trailing white spaces) in a multibyte code set where each of the characters and white-space characters consists of two bytes:

```
A1A2B1B2C1C2s1s2s1s2
```

Sometimes a string can contain both single-byte and multibyte white-space characters. In the following example, the string is composed of these elements: three single-byte characters (abc), a single-byte white-space character (s), a multibyte white-space character (s¹s²), two single-byte white-space characters (ss), and one multibyte white-space character (s¹s²):

```
abcss1s2ss1s2
```

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- On-line manuals
- Printed manuals
- Error message files
- Documentation notes, release notes, and machine notes
- Related reading

On-Line Manuals

A CD that contains Informix manuals in electronic format is provided with your Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see either the installation guide for your product or the installation insert that accompanies the documentation CD.

The documentation set that is provided on the CD describes Universal Server, its implementation of SQL, and its associated application-programming interfaces. For an overview of the manuals in the Universal Server documentation set, see [Getting Started with INFORMIX-Universal Server](#).

Printed Manuals

The Universal Server documentation set describes Universal Server, its implementation of SQL, and its associated application-programming interfaces. For an overview of the manuals in the Universal Server documentation set, see [Getting Started with INFORMIX-Universal Server](#).

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com.

Please provide the following information:

- The documentation that you need
- The quantity that you need
- Your name, address, and telephone number

Error Message Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. The **finderr** utility displays these error messages on the screen. For a detailed description of these error messages, see the Introduction to the [Informix Error Messages](#) manual.

UNIX

To read the error messages in the ASCII file, Informix provides scripts that let you display error messages on the screen (**finderr**) or print formatted error messages (**rofferr**). For a detailed description of these scripts, see the Introduction to the [Informix Error Message](#) manual. ♦

Windows

The FINDERR utility is a graphical tool. This utility has been created with Microsoft help facilities. For more information, see the “Error Message Files” section of the Introduction in your Informix documentation for Windows environments. ♦

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following on-line files, located in the `$INFORMIXDIR/release/en_us/0333` directory for UNIX or the `%INFORMIXDIR%\release\en_us\0333` for Windows NT, supplement the information in this manual.

On-Line File	Purpose
GLSDOC_9.1	The documentation-notes file describes features that are not covered in this manual or that have been modified since publication.
SERVERS_9.1	The release-notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
IUNIVERSAL_9.1	The machine-notes file describes any special actions that are required to configure and use Informix products on your computer. Machine notes are named for the product described.

Please examine these files because they contain vital information about application and performance issues.

Related Reading

For additional information about cultural and linguistic requirements around the world, and how computer systems handle those requirements, consult the following books on internationalization:

- *Developing and Localizing International Software* by Tom Madell, Clark Parsons, and John Abegg (Prentice-Hall, Inc., 1994)
- *Programming for the World: A Guide to Internationalization* by Sandra Martin O'Donnell (Prentice-Hall, Inc., 1994)

Compliance with Industry Standards

The GLS feature for Informix products is based on the X/Open XPG4 specifications. The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992, on INFORMIX-Universal Server. In addition, many features of Universal Server comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about corrections or clarifications that you would find useful. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
SCT Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send email, our address is:

`doc@informix.com`

Or send a facsimile to the Informix Technical Publications Department at:

415-926-6571

We appreciate your feedback.

GLS Fundamentals

What Is the GLS Feature?	1-3
GLS Support by Informix Products	1-5
Informix Database Servers	1-5
Informix Client Applications	1-6
Additional GLS Support	1-7
A GLS Locale	1-8
A Code Set	1-8
The Collation Order	1-10
End-User Formats	1-12
How Do You Set a GLS Locale?	1-18
The Default Locale	1-18
The Default Code Set	1-19
Default Date and Time End-User Formats	1-19
Default Numeric and Monetary End-User Formats	1-19
GLS Locale Names	1-20
Locales in the Client/Server Environment	1-22
The Client Locale	1-23
The Database Locale	1-26
The Server Locale	1-27
A Nondefault Locale	1-28
How Do Informix Products Use GLS Locales?	1-30
Supporting Non-ASCII Characters	1-30
Establishing a Database Connection	1-31
Sending the Client Locale	1-32
Verifying the Database Locale	1-33
Determining the Server-Processing Locale	1-34

Performing Code-Set Conversion.	1-40
What Is Code-Set Conversion?	1-41
When Is Code-Set Conversion Performed?	1-43
Which Data Is Converted?.	1-48
Locating Message Files	1-49
How Do You Customize Client End-User Formats?.	1-49
Customizing Date and Time End-User Formats.	1-50
Era-Based Date and Time Formats	1-51
Date and Time Precedence.	1-51
Customizing Monetary Values.	1-53

The Global Language Support (GLS) feature lets Informix products easily handle different languages, cultural conventions, and code sets for Asian, European, Latin American, and Middle Eastern countries.

This chapter introduces basic concepts and describes the GLS feature. It covers the following topics:

- What a GLS locale is and what information it provides
- How to establish a locale for Informix products
- How to customize end-user formats that a locale defines
- How a client application establishes a connection with a database
- How to use code-set conversion to convert character data between a client and a database, or a database and a file
- How to use localized message files with Informix products

What Is the GLS Feature?

In a database application, most of the tasks that the database server and the client application perform do not depend on the type of data that they handle. However, some portion of the tasks that the database server and client application perform are dependent on the data. For example, the database server must sort U.S. English data differently from Korean character data. The client application must display French currency differently than English currency.

If the Informix database server or the SQL flk product included the code to perform these data-dependent tasks, each would need to be written specially to handle a different set of culture-specific data. In fact, past Informix products have done that with the Asian Language Support (ALS) products for support of Asian (multibyte) data and the Native Language Support (NLS) feature for support of single-byte, non-English data.

With support for Global Language Support (GLS), Informix products no longer need to specify how to process this culture-specific information directly. Culture-specific information resides in a GLS locale. When an Informix product needs culture-specific information, it makes a call to the GLS library. The GLS library, in turn, accesses the GLS locale and returns the information to the Informix product. A GLS locale is a set of Informix files that bring together the information about data that is specific to a particular culture, language, or territory. In particular, a GLS locale provides the following information:

- The name of the code set that the application data uses
- The collation order to use for character data
- The format for different types of data to appear to end users

For more information on a GLS locale, see [page 1-8](#).

In addition, the GLS feature is a more portable way to support culture-specific information. Many operating systems provide support for non-English data. However, this support is usually in a form that is specific to the operating system. Not many standards yet exist for the format of culture-specific information. This lack of conformity means that if you move an application from one operating-system environment to another, you might need to change the way in which the application requests language support from the operating system. You might even find that the new operating-system environment does not provide the same aspect of language support that the initial environment provided.

Informix products support the GLS feature. Therefore, they can access culture-specific information regardless of the operating system under which they run. They can locate the locale information on any platform to which they are ported.

GLS Support by Informix Products

The Informix Version 9.1 release provides GLS support in the following types of products and utilities:

- INFORMIX-Universal Server
- Informix client applications, such as SQL API products and utilities (DB-Access, **onmode**, and so on)
- Informix GLS Application-Programming Interface (API)

The following sections outline the features that GLS support provides for each of these types of Informix products and utilities. For information about the GLS API, see [Chapter 4, “INFORMIX-Universal Server Features.”](#)

Informix Database Servers

With the GLS feature, INFORMIX-Universal Server, INFORMIX-OnLine Dynamic Server, and the INFORMIX-SE database server provide support for the following culture-specific features:

- Processing non-ASCII characters and strings
You can use non-ASCII characters to name user-specifiable database objects, such as tables, columns, views, statements, cursors, and stored procedures, and you can use a collation order that suits the local customs.
You can also use non-ASCII characters in many other contexts. For example, you can use them to evaluate the WHERE and ORDER BY clauses of your SELECT statements or to sort data in NCHAR and NVARCHAR columns. You can use GLS collation features without the modification of existing code.
- Evaluation of regular expressions
You can use non-ASCII characters in regular expression comparisons that involve NCHAR and NVARCHAR data.



- Translation of locale-specific values for dates, times, numeric data, and monetary data

You can use end-user formats that are particular to a country or culture outside the U.S. to specify date, time, numeric, and monetary values when they appear in literal strings. The database server can translate these formats to the appropriate internal database format.

- Accessibility of formerly incompatible character code sets

The client application can perform code-set conversion between convertible code sets to allow you to access and share data between databases and clients that have different code sets. For more information on code-set conversion, see [page 1-40](#).

Tip: For more information on how the database server provides support for the GLS feature, refer to [Chapter 3, “SQL Features,”](#) [Chapter 4, “INFORMIX-Universal Server Features,”](#) [Chapter 5, “INFORMIX-OnLine Dynamic Server Features,”](#) and [Chapter 6, “INFORMIX-SE Features.”](#)

Informix Client Applications

To the GLS feature, a *client application* can be either an Informix SQL API product, such as INFORMIX-ESQL/C or INFORMIX-ESQL/COBOL, or an Informix database server utility, such as DB-Access, **dbexport**, or **onmode**. The following Informix client applications provide support for the GLS feature:

- The DB-Access utility, which INFORMIX-Universal Server, INFORMIX-OnLine Dynamic Server, and INFORMIX-SE provide, allows user-specifiable database objects such as tables, columns, views, statements, cursors, and stored procedures to include non-ASCII characters and to be sorted according to localized collation rules.

For more information on SQL identifiers, see [“SQL Identifiers That Support Non-ASCII Characters” on page 4-17](#). For more general information about the DB-Access utility, refer to the [DB-Access User Manual](#).

- The SQL APIs allow host and indicator variable names as well as names of user-specifiable database objects such as tables, columns, views, statements, cursors, and stored procedures to include non-ASCII characters.

For more information, refer to [Chapter 7, “General SQL API Features.”](#)

- INFORMIX-Universal Server provides a new application programming interface, GLS API, to help you internationalize your programs. For more information on the GLS API, see [Chapter 4, “INFORMIX-Universal Server Features.”](#)
- Database server utilities, such as **dbexport** or **onmode**, allow many command-line arguments to include non-ASCII characters.

For more information, refer to the appropriate database server chapter: [Chapter 4, “INFORMIX-Universal Server Features,”](#) [Chapter 5, “INFORMIX-OnLine Dynamic Server Features,”](#) or [Chapter 6, “INFORMIX-SE Features.”](#)

Additional GLS Support

Informix products include the GLS locale files to support the default locale, U.S. English, and most non-Asian locales. (For more information on the default locale, see [page 1-18.](#)) If you do not find a locale to support your language and territory, you must install a Language Supplement for a particular language in addition to your Informix product.

A Language Supplement provides the locale files and error messages to support a particular language or set of languages. The International Language Supplement provides support for all non-Asian languages that Informix products support. You can obtain support for an Asian language (such as Korean, Japanese, or Chinese) with special add-on supplements.

For more information about the Language Supplements, contact your Informix sales representative. For more information about how to create customized message files, see [“Locating Message Files” on page 1-49.](#)

A GLS Locale

For data to be useful, it must be available in the format that the end user of the database application understands. This format depends on factors such as the language that the user speaks, and the culture or territory in which the user resides.

The GLS locale is set of Informix files that bring together the information about data that is specific to a particular culture, language, or territory. In particular, a GLS locale provides the following information:

- The name of the code set that the application data uses
- The collation order to use for character data
- The format for different types of data to appear to end users

This section describes each of these topics in more detail.

Tip: For information about the format of these GLS files, see [Appendix A](#).



A Code Set

A *character set* is one or more natural-language alphabets together with additional symbols for digits, punctuation, and diacritical marks. Each character set has at least one *code set*, which maps its characters to unique bit patterns. These bit patterns are called *code points*. ASCII, ISO8859-1, Microsoft 1252, and EBCDIC are examples of code sets that support the English language.

The number of unique characters in the language determines the amount of storage that each character requires in a code set. Because a single byte can store values in the range 0 to 255, it can uniquely identify 256 characters. Most Western languages have fewer than 256 characters and therefore have code sets made up of *single-byte characters*. When an application handles data in such code sets, it can assume that 1 byte stores 1 character.

The ASCII code set contains 128 characters. Therefore, the code point for each character requires 7 bits of a byte. These single-byte characters with code points in the range 0 to 128 are sometimes called ASCII or *7-bit characters*. The ASCII code set is a single-byte code set and is a subset of all code sets that Informix products support.

If a code set contains more than 128 characters, some of its characters have code points that must set the eighth bit of the byte. These non-ASCII characters might be either of the following types of characters:

- 8-bit characters

The 8-bit characters are single-byte characters whose code points are between 128 and 255. Examples from the ISO8859-1 or Microsoft 1252 code set include the non-English é, ñ, and ö characters. Only if the software is *8-bit clean* can it interpret these characters correctly. For more information on 8-bit characters and 8-bit clean software, see the description of the `GLS8BITFSYS` environment variable on [page 2-23](#).

- Multibyte characters

If a character set contains more than 256 characters, the code set must contain multibyte characters. A multibyte character might require from 2 to 4 bytes of storage. Many Asian languages contain 3,000 to 8,000 ideographic characters. Such languages have code sets made up of both single-byte and multibyte characters. These code sets are called multibyte code sets. Some characters in the Japanese SJIS code set are multibyte characters of 2 or 3 bytes. Applications that handle data in multibyte code sets cannot assume that 1 character takes only 1 byte of storage.



Tip: In this manual, the term *non-ASCII characters* applies to all characters with a code point greater than 127. Non-ASCII characters include both 8-bit and multibyte characters.

Informix products can support single-byte or multibyte code sets. For some examples of GLS locales that support non-ASCII characters, see “[Supporting Non-ASCII Characters](#)” on [page 1-30](#).



Tip: Throughout this manual, examples show how single-byte and multibyte characters appear. Because multibyte characters are usually ideographic (such as Japanese or Chinese characters), this manual does not use the actual multibyte characters. Instead, it uses ASCII characters to represent both single-byte and multibyte characters. For more information about how this manual represents multibyte and single-byte characters abstractly, see “[Character-Representation Conventions](#)” on [page 12](#) of the Introduction.

The Collation Order

Collation involves the sorting of character data that is either stored in a database or manipulated in a client application. The collation order affects the following tasks when you select from the database with the SQL SELECT statement:

- Logical predicates in the WHERE clause

```
SELECT * FROM tab1 WHERE col1 > 'bob'  
SELECT * FROM tab1 WHERE site BETWEEN 'abc' AND 'xyz'
```

- Sorted data that the ORDER BY clause creates

```
SELECT * FROM tab1 ORDER BY col1
```

- Comparisons in MATCHES and LIKE clauses

```
SELECT * FROM tab1 WHERE col1 MATCHES 'a1*'  
SELECT * FROM tab1 WHERE col1 LIKE 'dog'  
SELECT * FROM tab1 WHERE col1 MATCHES 'abc[a-z]'
```

For more information on how choice of a locale affects the SELECT statement, see [“Collation Order in SELECT Statements” on page 3-21](#).

Informix database servers support the following two methods of collation of character data:

- Code-set order
- Localized order

Code-Set Order

Code-set order refers to the bit-pattern order of characters within a code set. The order of the code points in the code set determines the sort order. For example, in the ASCII code set, A=65 and B=66. The character A always sorts before B because a code point of 65 is less than one of 66. However, because a=97 and M=77, the string *abc* sorts after *Me*, which is not always the preferred result.

The database server sorts data in CHAR, VARCHAR, and TEXT columns in code-set order. All code sets that Informix products support include the ASCII characters as the first 127 characters. Therefore, other characters in the code set have the code points 128 and greater. When the database server sorts data in a CHAR, VARCHAR, or TEXT column, it puts character strings that begin with ASCII characters before characters strings that begin with non-ASCII characters in the sorted results.

For an example of a data set in code-set order, see [Figure 3-2 on page 3-22](#).

Localized Order

Localized order refers to an order of the characters that relates to a real language. The locale defines the order of the characters in the localized order. For example, even though the character Å might have a code point of 133, the localized order could list this character after A and before B (A=65, Å=133, B=66). In this case, the string ÅB sorts after AC but before BD.

Tip: The *COLLATION* category of the locale file determines the localized order. For more information on the *COLLATION* category, see [page A-6](#).

The localized order can include *equivalent characters*, those characters that the database server is to consider as equivalent when it collates them. For example, if the locale defines uppercase and lowercase versions of a character as equivalent characters in the localized order, the database server considers the strings `Arizona`, `ARIZONA`, and `arizona` as equivalent and collates them together.

A localized order can also specify a certain type of collation order. It can define a telephone-book sorting order or a dictionary sort order. For example, a telephone book might require the following sort order:

```
Mabin
McDonald
MacDonald
Madden
```

A dictionary, however, might require the following sort order for these same names:

```
Mabin
Madden
MacDonald
McDonald
```

If the GLS locale defines a localized order, the database server sorts data in `NCHAR` and `NVARCHAR` columns in this localized order. For an example of a data set in localized order, see [Figure 3-3 on page 3-23](#).



Collation Support

The collation order that Informix database servers use depends on the data type of the database column. The following table summarizes these collation orders.

Data Types	Collation Order
CHAR, VARCHAR, TEXT	code-set order
NCHAR, NVARCHAR	localized order

The difference in collation order is the only distinction between the CHAR and NCHAR data types and the VARCHAR and NVARCHAR data types. For more information, see [“Using Character Data Types” on page 3-11](#).

NLS

Informix Native Language Support (NLS) database servers (before Version 7.2) use the same collation orders as Version 7.2 and later database servers: code-set order for CHAR and VARCHAR data and localized order for NCHAR and NVARCHAR data. ♦

ALS

Informix Asian Language Support (ALS) database servers use code-set order for CHAR and VARCHAR data; they do not support NCHAR and NVARCHAR data. ♦

If a locale does not define a localized order, the database server collates NCHAR and NVARCHAR data in code-set order.

End-User Formats

The *end-user format* is the format in which data appears within a client application when it is in literal strings or character variables. End-user formats are useful for data types whose format in the database is different from the format to which users are accustomed. In a database, the database server stores data for DATE, DATETIME, MONEY, and numeric data types in compact internal formats. For example, the database server stores a DATE value as an integer number of days since the date of December 31, 1899, so the date 03/19/96 is 35142. This internal format is not very intuitive.

Informix products support end-user formats so that a client application can use this more intuitive form instead of the internal format. Literal strings or character variables can appear in SQL statements as column values or as arguments of SQL API library functions.

An Informix product uses an end-user format when it encounters a string (a literal string or the value in a character variable) in the following contexts:

- When an Informix product *scans* a string, it uses an end-user format to determine how to interpret the string so that it can convert it to a numeric value.

For example, suppose DB-Access has the default locale (U.S. English) as its client locale. The literal date in the following INSERT statement uses the end-user format for dates that the default locale defines:

```
INSERT INTO mytab ( date1 ) VALUES ( '03/19/96' )
```

When the database server receives the data from the client application, the database server uses the end-user format to interpret this literal date so that it can convert it to the appropriate internal format (35142).

- When an Informix product *prints* a string, it uses an end-user format to determine how to format the numeric value as a string.

For example, suppose an ESQL/C client application has a French locale as its client locale, and this locale defines a date end-user format that formats dates as *dd/mm/yy*. The following **rdatestr()** function uses the end-user format for dates to obtain the value in the **datestr** character variable:

```
err = rdatestr(jdate, datestr);
```

The **rdatestr()** function uses the end-user format to determine how to format the internal format (35142) as a date string before it puts the value in the **datestr** variable. For more information about the effect of the GLS feature on SQL API library functions, see [“Enhanced ESQL Library Functions” on page 7-7](#).

A GLS locale defines end-user formats for the following types of data:

- Representation of currency notation and numeric format
You can use an end-user format that is particular to a country or culture outside the U.S. to specify monetary values.
- Representation of dates and times
You can specify date and time values in an end-user format that is particular to a country or culture outside the U.S.

The following sections describe each of these types of data in more detail.

Numeric and Monetary Formats

Numeric data is data from columns with the following data types: DECIMAL, INTEGER, SMALLINT, FLOAT, and SMALLFLOAT. Monetary data is data from a MONEY column. When an Informix product scans a string that contains monetary data, it uses the monetary end-user format to determine how to convert this string to the internal integer value for a MONEY column. When an Informix product prints a string that contains monetary data, it uses the monetary end-user format to determine how to format the internal integer value for a MONEY column as a string. In the same way, Informix products use the numeric end-user format to scan and print strings for the internal values of the numeric data types.



Important: *The end-user formats of the numeric and monetary data do not affect the internal format of the numeric or MONEY data types in the database. They only affect how the client application views the data.*

The end-user formats for numeric and monetary data specify the following characters and symbols:

- The *decimal-separator symbol*, often called the radix character, that separates the integral part of the numeric value from the fractional part
In the default locale, the period is the decimal separator (3.01); in a French or other European locale, the comma is the decimal separator (3,01).

- The *thousands-separator symbol* that appears between groups of digits in the integral part of the numeric value

In the default locale, the comma is the thousands separator (3,255); in a French locale, the space is the thousands separator (3 255).

- The number of digits to group between each appearance of a non-monetary thousands separator

For example, this information might specify that numbers always omit the separator at the millions position, which produces the following output: 1234,345.

- The characters that indicate positive and negative numbers

In addition to this numeric notation, monetary data also uses a *currency symbol* to identify the currency unit. A locale can define this symbol to appear at the front (\$100) or back (100FF) of the monetary value. In this manual, the combination of currency symbol, decimal separator, and thousands separator is called *currency notation*.



Tip: To customize the end-user format that the locale defines for monetary values, you can use the **DBMONEY** environment variable. For more information, see [“Customizing Monetary Values” on page 1-53](#).

The NUMERIC category of the locale file defines the end-user formats for numeric data. The MONETARY category of the locale file defines the end-user formats for monetary data. For more information on the NUMERIC and MONETARY categories, see [page A-7](#).

Date and Time Formats

Date data is for DATE or DATETIME columns. Time data is for a DATETIME column. When an Informix product scans a string that contains time data, it uses the time end-user format to determine how to convert this string to the internal integer value for a DATETIME column. When an Informix product prints a string that contains time data, it uses the time end-user format to determine how to format the internal integer value for a DATETIME column as a string. In the same way, Informix products use the date end-user format to scan and print strings for the internal values of the date data types.



Important: The end-user formats of the date and time data do not affect the internal format of the DATE or DATETIME data types in the database. They only affect how the client application views the data.

The end-user formats for date and time involve characters and symbols that format date and time values. This information includes the names and abbreviations for days of the week and months of the year. It also includes the commonly used representations for dates, time (12-hour and 24-hour), and DATETIME values.

The end-user formats can include the names of eras (as in the Japanese Imperial date system) and non-Gregorian calendars (such as the Arabic lunar calendar). For example, the Taiwan culture uses the Ming Guo year format in addition to the Julian calendar year. Ming Guo 0001 is equivalent to January 1, 1912 on the Julian calendar. For dates before 1912, Ming Guo years are negative. The Ming Guo year 0000 is undefined; any attempt to use it generates an error.

The following table shows some era-based dates.

Julian Year	Ming Guo Year	Remarks
1993	82	1993 - 1911 = 82
1912	01	1912 - 1911 = 01
1911	-01	1911 - 1912 = -01
1910	-02	1910 - 1912 = -02
1900	-12	1900 - 1912 = -12

Japanese Imperial-era dates are tied to the reign of the Japanese emperors. The following table shows Julian and Japanese era dates. It shows the Japanese era format in full, with abstract multibyte characters for the Japanese characters, and in an abbreviated form that uses romanized characters (gengo). The abbreviated form of the era uses the first letter of the English name for the Japanese era. For example, H represents the Heisei era.

Julian Date	Abstract Japanese Era (in full)	Japanese Era (gengo)
1868/09/08	A ¹ A ² B ¹ B ² 01/09/08	M01/09/08
1912/07/30	A ¹ A ² B ¹ B ² 45/07/30	M45/07/30
1912/07/31	A ¹ A ² B ¹ B ² 01/07/31	T01/07/31
1926/12/25	A ¹ A ² B ¹ B ² 15/12/25	T15/12/25
1926/12/26	A ¹ A ² B ¹ B ² 01/12/26	S01/12/26
1989/01/07	A ¹ A ² B ¹ B ² 64/01/07	S64/01/07
1989/01/08	A ¹ A ² B ¹ B ² 01/01/08	H01/01/08
⋮		
1995/01/01	A ¹ A ² B ¹ B ² 07/01/01	H07/01/01



Tip: In the preceding table, A¹A² and B¹B² indicate multibyte Japanese characters.

The TIME category of the locale file defines the end-user formats for date and time data. For more information on the TIME category, see [page A-8](#).



Tip: To customize the end-user formats that the locale defines for date and time values, you can use the GL_DATE and GL_DATETIME environment variables. For more information, see [“Customizing Date and Time End-User Formats” on page 1-50](#).

How Do You Set a GLS Locale?

In a client/server environment, both the database server and the client application must know which language the data is in to be able to process the application data correctly. The language affects database-server tasks such as sorting (collation) and comparison. It also affects client tasks such as end-user formats and alphabetic case conversion.

A GLS locale identifies the code set, the collation sequence, character classification (for comparisons), and end-user formats of monetary, numeric, date, and time data. For the database server and the client application to communicate successfully, you must establish the appropriate GLS locales for your environment.

This section describes the following topics:

- A default locale
- The format of a GLS locale name
- The locales that you must establish for Informix products
- How to establish a nondefault locale

The Default Locale

Informix products use U.S. English as the *default locale*. This locale specifies the following information:

- The U.S. English language and an English-language code set
- Standard U.S. formats for monetary, numeric, date, and time data

To use the default locale for your Informix database applications, you do not need to perform any special steps. However, if you want to use a customized version of U.S. English, or British English, or even another language, you must set environment variables to identify the appropriate locale. For information on how to specify a GLS locale, see [“A Nondefault Locale” on page 1-28](#).

UNIX

Windows

The Default Code Set

The *default code set* is the code set that the default locale supports. When you use the default locale, the default code set supports both the ASCII code set and some set of 8-bit characters. (For a chart of ASCII values, see the Relational Operator segment in the [Informix Guide to SQL: Syntax](#).) The name of the default code set is platform dependent.

The default code set on UNIX platforms is ISO8859-1. ♦

The default code set for Windows environments is Microsoft 1252. ♦

Default Date and Time End-User Formats

When you use the default locale, Informix products use the following end-user formats for date and time values:

- For DATE values: %m/%d/%iy
For more information on these formatting directives, see the description of `GL_DATE` on [page 2-25](#).
- For DATETIME values: %iY-%m-%d %H:%M:%S
For more information on these formatting directives, see the description of `GL_DATE` and `GL_DATETIME` on [pages 2-25](#) and [2-35](#), respectively.

For an introduction to date and time end-user formats, see “[Date and Time Formats](#)” on [page 1-15](#). For information about how to customize these end-user formats, see [page 1-50](#).

Default Numeric and Monetary End-User Formats

When you use the default locale, Informix products use the following end-user formats for numeric and monetary values:

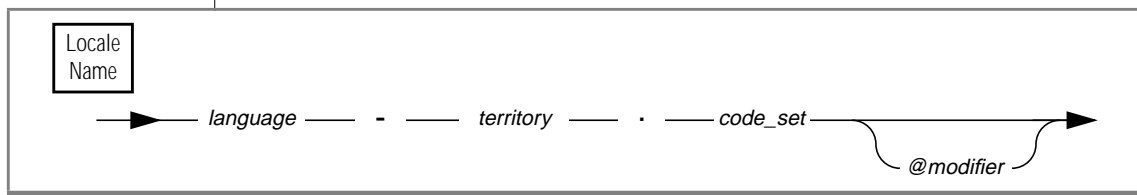
- The thousands separator is the comma (,).
- The decimal separator is the period (.)
- Three digits appear between each thousands separator.
- The positive and negative signs are the plus (+) and minus (-), respectively.

For monetary values, Informix products also use the currency symbol, which is the dollar sign (\$) and which appears in front of a monetary value.

For an introduction to numeric and monetary end-user formats, see [“Numeric and Monetary Formats” on page 1-14](#). For information about how to customize these end-user formats, see [page 1-53](#).

GLS Locale Names

A GLS locale name identifies the language, territory, and code set that you want your Informix product to use. This name has the following syntax.



code_set is the name of the code set that the locale supports.

language is the two-character name that represents the language for a specific locale.

modifier is an optional locale modifier that has a maximum of four alphanumeric characters. This specification modifies the cultural-convention settings that the *language* and *territory* settings imply. The modifier usually indicates a special type of localized order that the locale supports. For example, you can set *@modifier* to specify dictionary or telephone-book collation order.

territory is a two-character name that represents the cultural conventions. For example, *territory* might specify the Swiss version of the French, German, or Italian language.

The default locale, U.S. English, has the following locale name, where *en* indicates the English language, *us* indicates the United States territory, and *code_set* indicates the platform-specific name of the default code set:

en_us.code_set

UNIX

The name of the default locale on UNIX platforms is **en_us.8859-1**. The 8859-1 indicates the name of the default code set, the ISO8859-1 code set. ♦

Windows

The name of the default locale for Windows environments is **en_us.1252**. The 1252 indicates the name of the default code set, the Microsoft 1252 code set. ♦

In a locale name, you can specify the code set as either the code-set name or the condensed form of the code-set name. For example, the following locale names both identify the U.S. English locale with the ISO8859-1 code set:

- The locale name **en_us.8859-1** uses the code-set name to identify the ISO8859-1 code set.
- The locale name **en_us.0333** uses the condensed form of the code-set name to identify the ISO8859-1 code set.

For more information on the condensed form of a code-set name, see [page A-10](#).

The default locale has no locale modifier. However, Informix does provide the following locale that supports dictionary sort order for U.S. English, where *code_set* indicates the platform-specific name of the default code set:

```
en_us.code_set@dict
```

A sample nondefault locale for a French-Canadian locale follows:

```
fr_ca.8859-1
```

UNIX

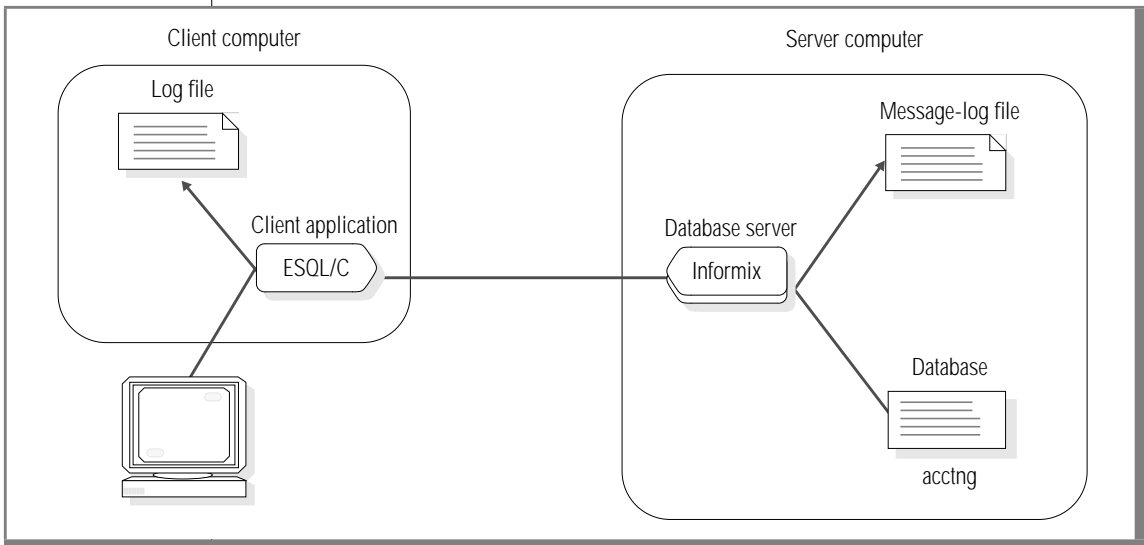
You can use the **glfiles** utility to generate a list of the GLS locales that are available on your UNIX system. For more information, see [“The glfiles Utility” on page A-21](#). ♦

An Informix product converts the locale name into a *GLS locale file*. A locale file is a runtime version of the locale information. The locale name must correspond to a GLS locale file in a subdirectory of the Informix installation directory (which the **INFORMIXDIR** environment variable indicates) called **gls**. For more information on GLS locale files, see [Appendix A, “Managing GLS Files.”](#)

Locales in the Client/Server Environment

When a database application runs in a client/server environment, the client application, database server, and database(s) might reside on different computers. Figure 1-1 shows a sample database server connection between an ESQL/C client application and the **acctng** database through an Informix database server (OnLine or Universal Server).

Figure 1-1
Example of a Client/Server Environment



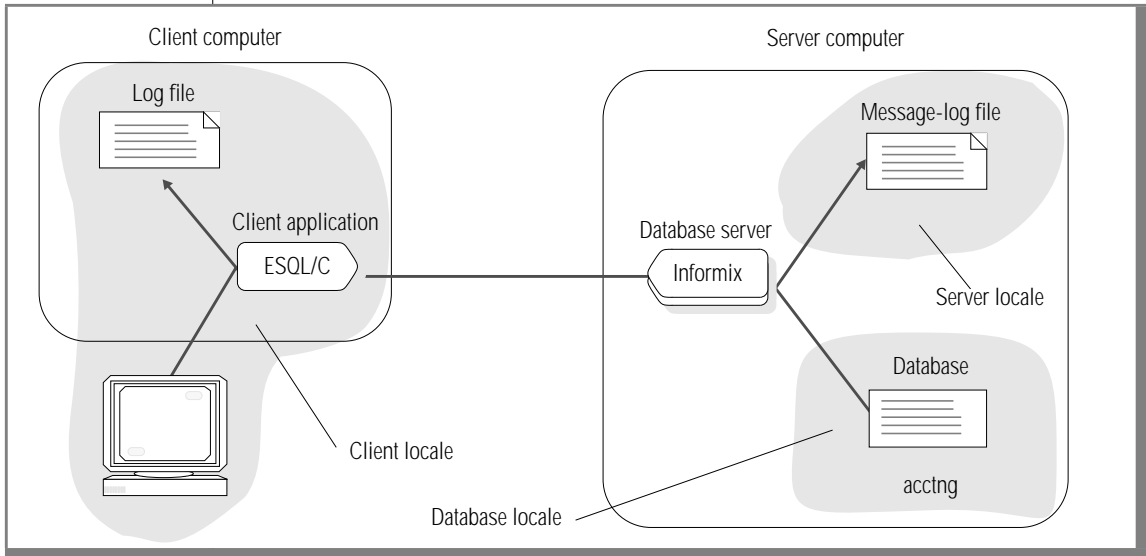
These computers might have different operating systems or different language support. To ensure that these three parts of the database application communicate locale information successfully, Informix products support the following locales:

- The *client locale* identifies the locale that the client application uses.
- The *database locale* identifies the locale of the data in a database.
- The *server locale* identifies the locale that the database server uses for its server-specific files.

Figure 1-2 shows the client locale, database locale, and server locale that the sample ESQL/C application (from [Figure 1-1 on page 1-22](#)) establishes.

Figure 1-2

The Client Locale, Database Locale, and Server Locale



When you set the same or compatible GLS locales for each of these locales, your client application is not dependent on how the operating system of each computer implements language-specific features.

The following sections describe each of these locales in more detail.

The Client Locale

The *client locale* specifies the language, territory, and code set that the client application uses to perform read and write (I/O) operations on the client computer. In a client application, I/O operations include reading a keyboard entry or a file for data to be sent to the database, and writing data that the database server retrieves from the database to the screen, a file, or a printer. In addition, an SQL API client uses the client locale for literal strings (end-user formats), embedded SQL statements, host variables, and data sent to or received from the database server with ESQL library functions in an ESQL source (.ec) file.



In the sample connection that Figure 1-2 shows, if the client locale is German with the Microsoft 1252 code set (**de_de.1252**), the German locale-specific information that the ESQL/C client application uses includes the following:

- Valid date end-user formats support the following format for the U.S. English date of Tuesday, 02/12/1996:

Mo., 12. Feb 1996

- Valid monetary end-user formats support the following format for the U.S. English amount of \$354,446.02:

DM354.446,02

Tip: To provide this information for the client locale, the locale file contains the following locale categories, *COLLATION*, *CTYPE*, *TIME*, *MONETARY*, and *NUMERIC*. For more information, see [“Locale Categories” on page A-4](#).

The database server uses the client locale as one part of the information to determine the server-processing locale for the current session. For more information, see [“Establishing a Database Connection” on page 1-31](#).

To determine the client locale, Informix client applications use environment variables set on the client computer. To obtain the localized order and end-user formats of the client locale, a client application uses the following precedence:

1. **DBDATE** and **DBTIME** environment variables for the end-user formats of date and time data and **DBMONEY** for the end-user format of monetary data (if one of these is set)
2. **GL_DATE** and **GL_DATETIME** environment variables for the end-user formats of date and time data (if one of these is set)
3. The information that the client locale defines (**CLIENT_LOCALE**, if it is set)
4. **LC_*** environment variables for the associated category of the client locale (if **DBNLS** is set on the client):
 - **LC_CTYPE** for the character-classification information (For more information about character classification, see the *CTYPE* category in [page A-6](#).)
 - **LC_COLLATE** for the localized order (For more information about the *COLLATE* category, see [page A-6](#).)

- ❑ **LC_NUMERIC** for numeric end-user formats (For more information about the NUMERIC category, see [page A-7.](#))
 - ❑ **LC_MONETARY** for monetary end-user formats (For more information about the MONETARY category, see [page A-7.](#))
 - ❑ **LC_TIME** for date and time end-user formats (For more information about the TIME category, see [page A-8.](#))
5. **LANG** environment variable (if, on the client, **DBNLS** is set, but the associated **LC_*** is not set) ♦
 6. The default locale (U.S. English)

Client applications that are based on Informix Version 9.1 products use the precedence of steps 2, 3, and 6 in the preceding list. You do not need to set the other environment variables for Version 9.1 client applications. Support for **DBDATE** and **DBTIME** provides backward compatibility for client applications that are based on earlier versions of Informix products. Informix recommends that you use **GL_DATE** and **GL_DATETIME** for new applications.

NLS

Support for **LANG** and the **LC_*** environment variables provides backward compatibility for client applications that are based on Informix NLS products. These applications use the precedence of steps , 4, 5, and 6 in the preceding list. You are not required to set **CLIENT_LOCALE** for NLS-based applications. However, **CLIENT_LOCALE** does provide certain advantages to your application. For more information, see the [Informix Migration Guide](#). ♦

ALS

Client applications that are based on Informix ALS products use the precedence of steps 3 and 6 in the preceding list. For more information about how to run ALS-based applications, see the [Informix Migration Guide](#). ♦

For information on the **CLIENT_LOCALE**, **DBDATE**, **DBTIME**, **DBMONEY**, **GL_DATE**, and **GL_DATETIME** environment variables, see their entries in [Chapter 2, “GLS Environment Variables.”](#)

The Database Locale

The *database locale* specifies the language, territory, and code set that the database server needs to interpret locale-sensitive data types (NCHAR and NVARCHAR) in a particular database correctly. The database locale determines the following information for the database:

- The code set whose characters are valid in any character column
- The code set whose characters are valid in the names of database objects such as databases, tables, columns, and views

For more information, see [“Naming Database Objects” on page 3-4](#).

- The localized order to collate data from any NCHAR and NVARCHAR columns

The database server uses the database locale as one part of the information to determine the server-processing locale for the current session. For more information, see [“Establishing a Database Connection” on page 1-31](#).

The database server also uses the database locale when it creates a new database. It stores a condensed version of the database locale in the **systables** catalog of the database.

NLS

Informix NLS databases (before Version 7.2) also store the locale-specific information in **systables**. However, the locale names are different from those in GLS databases. For information on how GLS database servers convert NLS databases, see the [Informix Migration Guide](#). ♦

ALS

Informix ALS databases store the locale-specification information somewhat differently. For information on how GLS database servers convert ALS databases, see the [Informix Migration Guide](#). ♦

When the database server stores the database locale information directly in the system catalog, it permanently attaches the locale to the database. In this way, the database server can always determine the locale that it needs to interpret the locale-sensitive data correctly. This information includes operations such as how to handle regular expressions, compare character strings, and ensure proper use of code sets.

In the sample connection shown in [Figure 1-2 on page 1-23](#), the database server references the database locale when the client application requests sorted information for an NCHAR column in the **acctng** database. If the database locale is German with the Microsoft 1252 code set (**de_de.1252**), the database server uses a localized order that sorts accented characters, such as ö, after their unaccented counterparts. This order means that the string öff sorts after ord but before pre.

The client application uses the database locale (as set on the client computer) to determine whether to perform code-set conversion. For more information, see [“Performing Code-Set Conversion” on page 1-40](#).

The Server Locale

The *server locale* specifies the language, territory, and code set that the database server uses to perform read and write (I/O) operations on the server computer (the computer on which the database server runs). These I/O operations include reading or writing the following files:

- Diagnostic files that the database server generates to provide additional diagnostic information
- Log files that the database server generates to record events
- Explain file, **sqexplain.out**, that the SQL statement SET EXPLAIN generates

However, the database server does not use the server locale to write files that are in an Informix-proprietary format (database and table files). For a more detailed description of the files that the database server writes in the server locale, refer to [Chapter 4, “INFORMIX-Universal Server Features,”](#) [Chapter 5, “INFORMIX-OnLine Dynamic Server Features,”](#) and [Chapter 6, “INFORMIX-SE Features.”](#)

In the sample connection that [Figure 1-2](#) shows, the Informix database server uses the server locale to determine the code set to use when it writes a message-log file.



Tip: *The database server is the only Informix product that needs to know the server locale. Any database server utilities that you run on the server computer use the client locale to read from and write to files and the database locale (on the server computer) to access databases that are set on the server computer.*

A Nondefault Locale

By default, Informix products use the U.S. English locale. However, Informix products support many other locales. To use a nondefault locale, you must set the following locale environment variables:

- Set the **CLIENT_LOCALE** environment variable to specify a nondefault client locale.

If you do not set **CLIENT_LOCALE**, the client locale is the default locale, U.S. English.

- Set **DB_LOCALE** on each client computer to specify a nondefault database locale for a client application to use when it connects to a database.

If you do not set **DB_LOCALE**, on the client computer, the client application sets the database locale to the client locale. This default value prevents the client application from the need to perform code-set conversion.

You might also want to set **DB_LOCALE** on the server computer so that the database server can perform operations such as the creation of databases (when the client does not specify its own **DB_LOCALE**).

- Set the **SERVER_LOCALE** environment variable to specify a nondefault server locale.

If you do not set **SERVER_LOCALE**, the server locale is the default locale, U.S. English.

To set these locale environment variables, assign them the name of a GLS locale (see [page 1-20](#)).

You set environment variables with the appropriate shell command (such as **setenv** for the C shell).

For example, to set the client locale to Japanese with the SJIS character set on a UNIX system, you might enter the following command on the client computer at the shell prompt:

```
setenv CLIENT_LOCALE ja_jp.sjis
```

UNIX

Windows

You can use the **glfiles** utility to obtain a list of GLS locales that your Informix product supports. For more information, see [“The glfiles Utility” on page A-21](#). ♦

You set environment variables in the Registry (with the SETNET32 utility) or in the **InetLogin** structure of an SQL API program. For more information on SETNET32, see the *Installation and Configuration Guide for Microsoft Windows Environments*. For more information on **InetLogin**, see the Microsoft Windows supplement for your SQL API documentation.

You can examine the %**INFORMIXDIR%****gls\lcX** directory to determine the GLS locales that your Informix product supports. For more information on this directory, see [Appendix A](#). ♦

When you want to access a database locale with a nondefault locale, the client and database locales on your client computer must support this nondefault locale. Make sure that these two locales are the same or that their code sets are convertible. For more information on convertible code sets, see [“Performing Code-Set Conversion” on page 1-40](#).

For example, to access a database with a Japanese SJIS locale, set both the **DB_LOCALE** and **CLIENT_LOCALE** environment variables to the **ja_jp.sjis** locale name. If you set **DB_LOCALE** but do not set **CLIENT_LOCALE**, the client application returns an error because it cannot set up code-set conversion between SJIS (the database code set) and the default code set (the code set of the default locale).

When a client application requests a connection, the database server uses information in the client, database, and server locales to create the server-processing locale. For more information, see [“Establishing a Database Connection” on page 1-31](#). For information on the syntax of the **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables, refer to those entries in [Chapter 2, “GLS Environment Variables.”](#)

How Do Informix Products Use GLS Locales?

Informix products use GLS locales for the following tasks:

- When a client application requests a connection, the database server uses the client and database locales to determine if these locales are compatible.
- When a client application first begins execution, it compares the client and database locales to determine if it needs to perform code-set conversion.
- All Informix products that display product-specific messages look in a directory specific to the client locale to locate these messages.

Supporting Non-ASCII Characters

An Informix product determines which code set it uses from the name of a GLS locale. Informix provides locales that support both single-byte and multibyte code sets. All code sets that Informix supports define the ASCII characters. Most also support additional non-ASCII characters (8-bit or multibyte characters). For more information on code sets and non-ASCII characters, see [page 1-8](#).

The following types of GLS locales are examples of locales that contain non-ASCII characters in their code sets:

- The default locale supports the default code set, which contains 8-bit characters for non-English characters such as é, ñ, and ö.
The name of the default code set depends on the platform on which your Informix product is installed. For more information on the default code set, [page 1-19](#).
- Many nondefault locales support the default code set.
Nondefault locales that support the UNIX default code set, ISO8859-1, include British English (**en_gb.8859-1**), French (**fr_fr.8859-1**), Spanish (**es_es.8859-1**), and German (**de_de.8859-1**). ♦
- Other nondefault locales, such as Japanese SJIS (**ja_jp.sjis**), Korean (**ko_kr.ksc**), and Chinese (**zh_cn.gb**), contain multibyte code sets.

The following table summarizes contexts in which you can use non-ASCII characters, including multibyte characters.

Context for Non-ASCII Characters	For More Information
SQL data	Chapter 3, “SQL Features”
Universal Server data	Chapter 4, “INFORMIX-Universal Server Features”
OnLine data	Chapter 5, “INFORMIX-OnLine Dynamic Server Features”
SE data	Chapter 6, “INFORMIX-SE Features”
SQL API data	Chapter 7, “General SQL API Features”

However, for an Informix product to support non-ASCII characters, it must use a locale that supports a code set with these same non-ASCII characters.

Establishing a Database Connection

When a client application requests a connection to a database, the database server uses GLS locales to perform the following steps:

1. Examine the client locale information that the client passes.
2. Verify that it can establish a connection between the client application and the database that it requested.
3. Determine the server-processing locale, which the database server uses to handle locale-specific information for the connection.

Sending the Client Locale

When the client application requests a connection, it sends the following environment variables from the client locale to the database server:

- **Locale information**

- **CLIENT_LOCALE and DB_LOCALE**

If **CLIENT_LOCALE** is not set (and **DBNLS** is *not* set), the client sets it to the default locale. If **DB_LOCALE** is not set (and **DBNLS** is *not* set), the client does not send a **DB_LOCALE** value to the database server.

- **DBNLS and LC_***

The client application sends the **LC_*** environment variables *only* if **DBNLS** is set. If **DBNLS** is set but the **LC_*** variables are not, the client sets the appropriate **LC_*** values, based on the **LANG** environment variable, and sends these **LC_*** values to the database server. The client application does *not* send the value of the **LANG** environment variable to the database server. ♦

- **DBCODESET**

A Version 5.x ALS client application can send **DBCODESET** to a Version 7.2 database server; the database server converts the **DBCODESET** value to a valid locale name and uses it as **DB_LOCALE**. A Version 6.x ALS client application sends the **CLIENT_LOCALE** and **DB_LOCALE** environment variables. ♦

- **User-customized end-user formats**

- **Date and time end-user formats: GL_DATE and GL_DATETIME or DBDATE and DBTIME**

- **Monetary end-user formats: DBMONEY**

If you do not set any of these environment variables, the client application does not send them to the database server, and the database server uses the end-user formats that the client locale defines.

The database server uses this information to verify the database locale and to establish the server-processing locale.

NLS

ALS

Verifying the Database Locale

To open an existing database, the client application must correctly identify the database locale for the database(s) that it needs to access. The database server verifies the database locale from the information that the client sends when it requests a database server connection. To perform this verification, the database server compares the following two database locales:

- The database locale from the **DB_LOCALE** environment variable that the client application sends
For more information, see [“Sending the Client Locale” on page 1-32](#).
- The database locale that is stored in the system catalog of the database that the client application requests
For more information, see [“The Database Locale” on page 1-26](#).

Two database locales match if their language, territory, code set, and any optional locale modifiers are the same. If these database locales do not match, the database server performs the following actions:

- It sets the SQLWARN7 warning flag in the SQL Communications Area (SQLCA structure). For more information on how to check the SQLWARN7 warning flag, see [“Checking for Connection Warnings” on page 7-24](#).
- It uses the database locale that is stored in the system catalog of the requested database as the database locale.



Warning: Check for the SQLWARN7 warning flag after your ESQL client application requests a connection. If the two database locales do not match, the client application might incorrectly interpret data that it retrieves from the database, or the database server might incorrectly interpret data that it receives from the client. If you proceed with such a connection, it is your responsibility to understand the format of the data that is being exchanged.

For example, suppose your client application uses **CLIENT_LOCALE** set to **en_us.1252** and **DB_LOCALE** set to **en_us.8859-1** and then opens a database with the French **fr_fr.8859-1** locale. The database server sets SQLWARN7 because the languages and territories of the two locales are different. The database server then uses the locale of the database (**fr_fr.8859-1**) for the localized order of the data.

NLS

However, your application might choose to use this connection. It might be acceptable for the application to receive the NCHAR and NVARCHAR data that is sorted in a French localized order. Any code-set conversion that the client application performs would still be valid because both database locales support the ISO8859-1 code set. (For more information on code-set conversion, see [page 1-40](#).)

Instead, if the application opens a database with the Japanese SJIS (**ja_jp.sjis**) locale, the database server sets the SQLWARN7 flag because the language, territory, *and* code sets differ. The database server then uses the **ja_jp.sjis** locale for the localized order of the data. Your application would probably *not* continue with this connection. Because the code sets are different, the client application would not perform code-set conversion correctly.

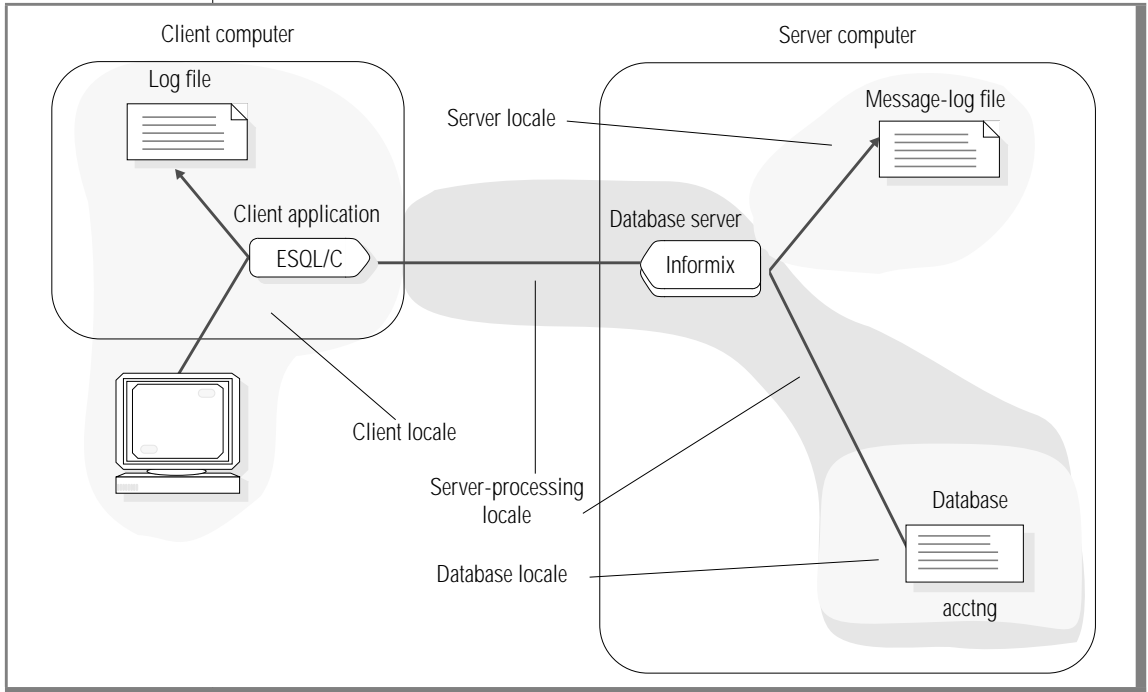
If **DBNLS** is set to one (1), the database server does not allow the client to connect to a database if the database locale that the client sends does not match the database locale of the NLS database that the client requested. ♦

Determining the Server-Processing Locale

The database server uses the *server-processing locale* to obtain locale information for its own internal sessions and for any connections. When the database server begins execution, it initializes the server-processing locale to the default locale. When a client application requests a connection, the database server must redetermine the server-processing locale to include the client and database locales. The database server uses the server-processing locale to obtain locale information that it needs when it transfers data between the client and database.

Once the Informix database server verifies the database locale (see [page 1-33](#)), it uses a precedence of environment variables from the client and database locales to set the server-processing locale. Figure 1-3 shows the relationship between the client locale, database locale, server locale, and server-processing locale.

Figure 1-3
The Server-Processing Locale



The database server obtains the following information from the server-processing locale:

- **Locale information for the database**

This database information includes the localized order and code set for data in columns with the locale-specific data types (NCHAR and NVARCHAR). The database server obtains this information from the name of the database locale that it has just verified (see [page 1-33](#)).

- **Locale information for client-application data**

This client-application information is the end-user formats for date, time, and monetary data. The database server obtains this information from the client application when the client requests a connection (see [page 1-32](#)).



Tip: The database server uses the server locale for read and write operations on its own operating-system files. For information about operating-system files, see “[Using the Server Locale](#)” on [page 4-3](#) for Universal Server, [page 5-3](#) for OnLine, and [page 6-3](#) for SE.

Locale Information for the Database

The database server must know how to interpret the data in any columns with the locale-specific data types, NCHAR and NVARCHAR. To handle this locale-specific data correctly, the database server must know the localized order for the collation of the data and the code set of the data. In addition, the database server uses the code set of the database locale as the code set of the server-processing locale. The database server might have to perform code-set conversion between the code sets of the server-processing locale and the server locale. For more information, see “[Performing Code-Set Conversion](#)” on [page 1-40](#).

The database server uses the following precedence to determine this database information:

1. The locale that the database server uses to determine the database information for the server-processing locale depends on the state of the database to which the client application requests a connection, as follows:
 - a. For a connection to an *existing* database, the database server uses the database information from the database locale that it obtains when it verifies the database locale (see [page 1-33](#)). If the client application does not send **DB_LOCALE**, the database server uses the **DB_LOCALE** that is set on the server computer.
 - b. For a *new* database, the database server uses the **DB_LOCALE**, which the client application has sent (see [page 1-32](#)).
2. **LC_COLLATE** and **LC_CTYPE** environment variables that the client application passes (if **DBNLS** is set on the client computer) ♦
3. The locale that the **DB_LOCALE** environment variable on the server computer indicates
4. **LC_COLLATE** and **LC_CTYPE** environment variables on the server computer (if **DBNLS** is set on the client computer)
5. **LANG** environment variable on the server computer (if **DBNLS** is set on the client) ♦
6. **DBCODESET** environment variable that the client application passes
7. **DBCODESET** environment variable on the server computer (if **DBCODESET** is *not* set on the client computer) ♦
8. The default locale (U.S. English)

Informix Version 9.1 database servers use the precedence of steps 3 and 8 in the preceding list to obtain the database information for the server-processing locale. You are not required to set the other environment variables.

Support for **LANG** and the **LC_*** environment variables provides backward compatibility for client applications that are based on Informix NLS products. These applications use the precedence of steps 2, 4, 5, and 8 in the preceding list. You are not required to set **CLIENT_LOCALE** for NLS-based applications. However, **CLIENT_LOCALE** does provide certain advantages to your application. For more information, see the [Informix Migration Guide](#). ♦

NLS

NLS

ALS

NLS

ALS



Support for the **DBCODASET** environment variable provides backward compatibility for client applications that are based on Informix Version 5.x ALS products. These applications use the precedence of steps , 3, 6, 7, and 8 in the preceding list. For more information about how to run ALS-based applications, see the [Informix Migration Guide](#). ♦

***Tip:** The precedence rules apply to how the database server determines both the **COLLATION** category and the **CTYPE** category of the server-processing locale. For more information on these locale categories, see [page A-4](#).*

For more information on how the database server obtains these environment variables, see “[Sending the Client Locale](#)” on [page 1-32](#).

If the client application makes another request to open a database, the database server must reestablish the database information for the server-processing locale, as follows:

1. Reverify the database locale by comparing the database locale in the database to be opened with the value of the **DB_LOCALE** environment variable from the client application (see [page 1-33](#)).
2. Reestablish the server-processing locale with the newly verified database locale (from the preceding step).

For example, suppose that your client application has **DB_LOCALE** set to **en_us.8859-1** (U.S. English with the ISO8859-1 code set). The client application then opens a database with the U.S. English (**en_us.8859-1**) locale, and the database server establishes a server-processing locale with **en_us.8859-1** as the locale that defines the database information.

If the client application now closes the U.S. English database and opens another database, one with the French (**fr_fr.8859-1**) locale, the database server must reestablish the server-processing locale. The database server sets the **SQLWARN7** flag to indicate that the two locales are different. However, your client application might choose to use this connection because both these locales support the ISO8859-1 code set. If the client application opens a database with the Japanese SJIS (**ja_jp.sjis**) locale instead of one with a French locale, your client application would probably not continue with this connection because the locales are too different.

Locale Information For the Client Application

The database server must know how to interpret the end-user formats when they appear in monetary, date, or time data that the client application sends. It must also convert data from the database to any appropriate end-user format before it sends this data to the client application. (For more information about end-user formats, see [page 1-12](#).)

The database server uses the following precedence to determine this client-application information:

1. **DBDATE** and **DBTIME** environment variables for the date and time end-user formats and **DBMONEY** for the monetary end-user formats (if one of these is set on the client)
2. **GL_DATE** and **GL_DATETIME** environment variables (if one of these is set on the client) for the date and time end-user formats
3. The locale that the **CLIENT_LOCALE** environment variable from the client application indicates (see [page 1-32](#))
4. If **DBNLS** is set on the client computer (for an NLS-based application):
 - a. **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME** environment variables that the client application passes to define end-user formats of monetary, numeric, and date/time data, respectively
 - b. **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME** environment variables on the server computer for the end-user formats of monetary, numeric, and date/time data, respectively
 - c. **LANG** environment variable on the server computer ♦
5. **DBCODESET** environment variable that the client passes ♦
6. The default locale (U.S. English)

Informix Version 9.1 database servers use the precedence of steps 2, 3, and 6 in the preceding list to set the end-user formats for the server-processing locale. You are not required to set the other environment variables. Support for **DBDATE** and **DBTIME** provides backward compatibility for client applications that are based on earlier versions of Informix products. Informix recommends that you use **GL_DATE** and **GL_DATETIME** for new applications.

NLS

ALS

NLS

Support for **LANG** and the **LC_*** environment variables provides backward compatibility for client applications that are based on Informix NLS products. If **DBNLS** is set on the client, these applications use the precedence of steps 4, 5, and 6 in the preceding list to set the end-user formats for the server-processing locale. You are not required to set **CLIENT_LOCALE** for NLS-based applications. However, **CLIENT_LOCALE** does provide certain advantages to your application. For more information, see the [Informix Migration Guide](#). ♦

ALS

Support for the **DBCODASET** environment variable provides backward compatibility for client applications that are based on Informix Version 5.x ALS products. These applications use the precedence of steps 3, 5, and 6 in the preceding list. For more information about how to run ALS-based applications, see the [Informix Migration Guide](#). ♦



***Tip:** The precedence rules apply to how the database server determines the **NUMERIC MONETARY**, **TIME**, and the **MESSAGES** categories of the server-processing locale. For more information on these locale categories, see [page A-4](#).*

The client application passes the **DBCODASET**, **DBDATE**, **DBMONEY**, **DBTIME**, **GL_DATE**, **GL_DATETIME**, and **LC_*** environment variables (if they are set) to the database server. It also passes the **CLIENT_LOCALE** and **DB_LOCALE** environment variables. For more information, see [“Sending the Client Locale” on page 1-32](#).

Performing Code-Set Conversion

Informix products use GLS locales to perform code-set conversion. Both an Informix client application and a database server might perform code-set conversion. This section provides the following information about how Informix products support code-set conversion:

- What code-set conversion is
- When code-set conversion is performed
- Which data is converted

What Is Code-Set Conversion?

You specify a code set as part of the GLS locale (see [page 1-20](#)). At runtime, Informix products adhere to the following rules to determine which code sets to use:

- The client application uses the *client code set*, which the **CLIENT_LOCALE** environment variable specifies, to write all files on the client computer and to interact with all client I/O devices.
- The database server uses the *server code set*, which the **SERVER_LOCALE** environment variable specifies, to write files (such as debug and warning files).
- The database server uses the *database code set*, which the **DB_LOCALE** environment variable specifies, to transfer data to and from the database.

In a client/server environment, character data might need to be converted from one code set to another if the client or server computer uses different code sets to represent the same characters. The conversion of character data from one code set (the source code set) to another (the target code set) is called *code-set conversion*. Without code-set conversion, one computer cannot correctly process or display character data that originates on the other (when the two computers use different code sets).

Code-set conversion does not provide either of the following capabilities:

- Code-set conversion is not a semantic translation.
It does not convert between words in different languages. For example, it does not convert from the English word *yes* to the French word *oui*. It only ensures that each character retains its meaning when it is processed or printed, regardless of how it is encoded.
- Code-set conversion does not create a character in the target code set if it exists only in the source code set.
For example, if the character *â* is passed to a target computer whose code set does not contain that character, the target computer can never process or print the character exactly.

For each character in the source code set, a corresponding character in the target code set should exist. However, if the source code set contains characters that are not in the target code set, the conversion must then define how to map these mismatched characters to the target code set. (Absence of a mapping between a character in the source and target code sets is often called a *lossy* error.) If all characters in the source code set exist in the target code set, mismatch handling does not apply.

A code-set conversion uses one of the following four methods to handle mismatched characters:

- Round-trip conversion

This method maps each mismatched character to a unique character in the target code set so that the return mapping maps the original character back to itself. This method guarantees that a two-way conversion results in no loss of information; however, data that is converted just one way might prevent correct processing or printing on the target computer.

- Substitution conversion

This method maps all mismatched characters to one character in the target code set that highlights mismatched characters. This method guarantees that a one-way conversion clearly shows the mismatched characters; however, a two-way conversion results in loss of information if mismatched characters are present.

- Graphical-replacement conversion

This method maps each mismatched character to a character in the target code set that looks similar to the source character. (This method includes the mapping of one-character ligatures to their two-character equivalents and vice versa.) This method tries to make printing of mismatched data more accurate on the target computer, but it most likely confuses the processing of this data on the target computer.

- A hybrid of two or three of the preceding conversion methods

Tip: Each code-set-conversion source file (.cv) indicates how the associated conversion handles mismatched characters. For information on code-set-conversion files, see [Appendix A, “Managing GLS Files.”](#)



When Is Code-Set Conversion Performed?

An application needs to use code-set conversion only when the two code sets (client and server-processing locale, or server-processing locale and server) are different. The following situations are possible causes of code sets that differ:

- Different operating systems might encode the same characters in different ways.

For example, the code for the character â (a-circumflex) in Windows Code Page 1252 (a common Microsoft code set) is hexadecimal 0xE2. In IBM Coded Character Set Identifier (CCSID) 437 (a common IBM UNIX code set), the code is hexadecimal 0x83. If the code for â on the client is sent unchanged to the IBM UNIX computer, it prints as the Greek character γ (gamma). This action occurs because the code for γ is hexadecimal 0xE2 on the IBM UNIX computer.

- One language can have several code sets. Each might represent a subset of the language.

For example, the code sets **cdlc** and **big5** are both internal representations of a subset of the Chinese language. However, these subsets consist of different numbers of Chinese characters.



***Tip:** The IBM CCSID code-set numbers are a system of 16-bit numbers that uniquely identify the coded graphic character representations. Informix products support the CCSID numbering system. For more information, see [Appendix A, “Managing GLS Files.”](#)*

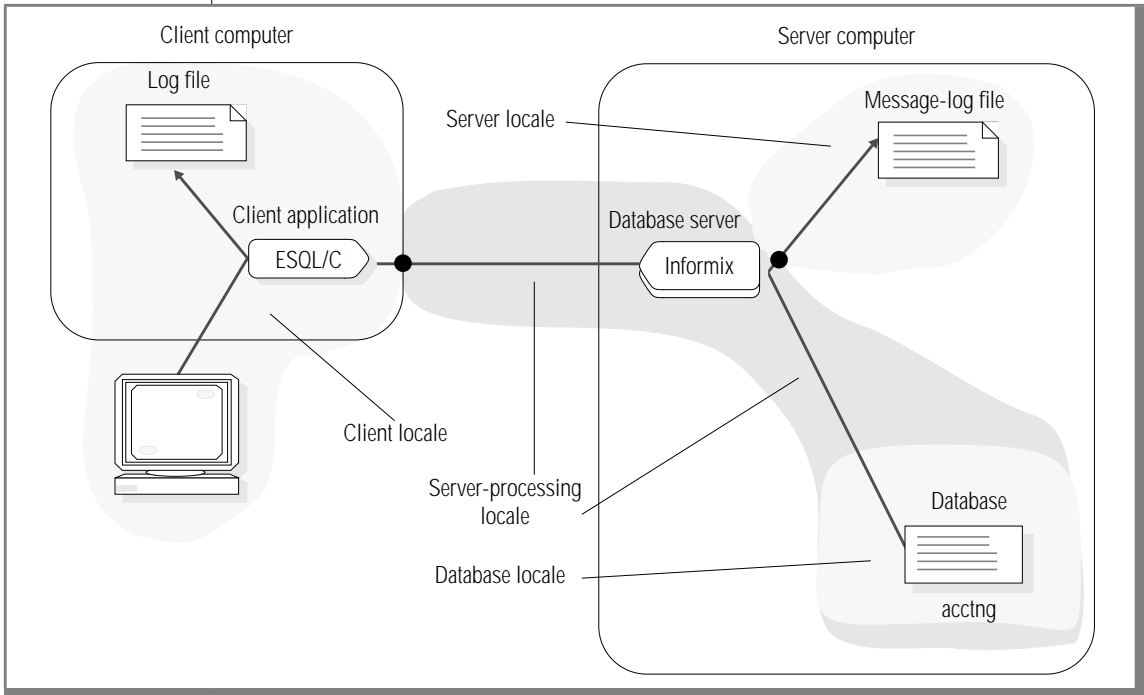
If a code-set conversion is required when data goes from computer A to computer B, it is also required when the data goes from computer B to computer A.

In the client/server environment, the following situations might require code-set conversion:

- If the client locale and database locale specify different code sets, the client application performs code-set conversion so that the server computer is not loaded with this type of processing.
- If the server locale and server-processing locale specify different code sets, the database server performs code-set conversion when it writes to and reads from operating-system files such as log files.

In Figure 1-4, the black dots indicate the two points in a client/server environment at which code-set conversion might occur.

Figure 1-4
Points of GLS Code-Set Conversion



In the sample connection that Figure 1-4 shows, the ESQL/C client application performs code-set conversion on the data that it sends to and receives from the database server if the client and database code sets are convertible. The Informix database server also performs code-set conversion when it writes to a message-log file if the code sets of the server locale and server-processing locale are convertible.

NLS

Informix products provide operating-system locales for backward compatibility with NLS databases. (For information on how to list operating-system locales, see [Appendix A, “Managing GLS Files.”](#)) If your NLS-based client application (**DBNLS** is set) uses an operating-system locale for either the client or the database locale, your NLS-based application does not automatically perform code-set conversion. For code-set conversion to occur, you must set the **DBAPICODE** environment variable. For more information, see the [Informix Migration Guide](#). ♦

Client Application Code-Set Conversion

An Informix client application automatically performs code-set conversion between the client and database code sets when the following conditions are true:

- The code sets that the client and database locales support do not match.
- A valid object code-set-conversion exists for the conversion between the client and database code sets.

When the client application begins execution, it compares the names of the client and database locales to determine whether to perform code-set conversion. If the **CLIENT_LOCALE** and **DB_LOCALE** environment variables are set, the client application uses these locale names to determine the client and database code sets, respectively. If **CLIENT_LOCALE** is not set (and **DBNLS** is *not* set), the client application assumes that the client locale is the default locale. If **DB_LOCALE** is not set (and **DBNLS** is *not* set), the client application assumes that the database locale is the same as the client locale (the value of **CLIENT_LOCALE**).

NLS

If **DBNLS** is set on the client application, the client application is to connect to an NLS database server. This client application still performs code-set conversion between the code sets that the **CLIENT_LOCALE** and **DB_LOCALE** environment variables indicate. However, if you set **DB_LOCALE** and *not* **CLIENT_LOCALE**, the client application performs code-set conversion between the client code set, which the **LC_CTYPE** environment variable indicates (or **LANG**, if **LC_CTYPE** is not set), and the database code set, which **DB_LOCALE** indicates. ♦

ALS

If the Version 9.1 client application is to connect to a Version 6.0 ALS database server, it sets **CLIENT_LOCALE** and **DB_LOCALE** to indicate the client and database locales. This client application still performs code-set conversion between the code sets that the **CLIENT_LOCALE** and **DB_LOCALE** environment variables indicate. If the Version 9.1 client application is to connect to a Version 5.x ALS database server, you must set **CLIENT_LOCALE** and **DB_LOCALE** to indicate the client and database code sets. In this case, the Version 9.1 client application performs code-set conversion between these two code sets. ♦

If the client and database code sets are the same, then no code-set conversion is needed. However, if the code sets do not match, the client application must determine whether the two code sets are *convertible*. Two code sets are convertible if the client can locate the associated code-set-conversion files. These code-set-conversion files must exist on the client computer.

UNIX

You can use the **glfiles** utility to obtain a list of code-set conversions that your Informix product supports. For more information, see [“The glfiles Utility” on page A-21](#). ♦

Windows

You can examine the **%INFORMIXDIR%\gls\cvY** directory to determine the GLS code-set conversions that your Informix product supports. For more information on this directory, see [Appendix A](#). ♦

If no code-set-conversion files exist, the client application generates a run-time error when it starts up to indicate that the code sets are incompatible. If these code-set-conversion files exist, the client application automatically performs code-set conversion when it sends data to or receives data from the database server.

When a client application performs code-set conversion, it makes the following assumptions:

- All database data within the client application is handled in the client code set.
- All databases that the client application accesses on a single database server use the *same* database locale, territory, and code set. When the client application opens a different database, it does *not* recheck the database locale to determine if the code set has changed.



Warning: Check for the `SQLWARN7` warning flag after each request for a connection. If the two database locales do not match, the client application might be performing code-set conversion incorrectly. The client application continues to perform any code-set conversion based on the code set that **DB_LOCALE** supports. If you proceed with such a connection, it is your responsibility to understand the format of the data that is being exchanged.

For example, suppose your client application has **CLIENT_LOCALE** set to **en_us.1252** and **DB_LOCALE** set to **en_us.8859-1**. The client application determines that it must perform code-set conversion between the Microsoft 1252 code set (in the client locale) and the ISO8859-1 code set (in the database locale). The client application then opens a database with the French **fr_fr.8859-1** locale. The database server sets `SQLWARN7` and uses the locale of the database (**fr_fr.8859-1**) for the localized order of the data. However, your application might choose to use this connection. The code-set conversion that the client application performs would still be valid because both database locales support the ISO8859-1 code set.

Instead, if the application opens a database with the Japanese SJIS (**ja_jp.sjis**) locale, the database server sets the `SQLWARN7` flag and uses the **ja_jp.sjis** locale for the localized order of the data. Your application would probably *not* continue with this connection. When the client application started, it determined that code-set conversion was required between the Microsoft 1252 and ISO8859-1 code sets. The client application performs this code-set conversion until it terminates. When you open a database with the **ja_jp.sjis**, the client application would perform code-set conversion incorrectly. It would continue to convert between Microsoft 1252 and ISO8859-1 instead of between Microsoft 1252 and Japanese SJIS. This situation could lead to serious corruption of data.

Database Server Code-Set Conversion

An Informix database server automatically performs code-set conversion between the code sets of the server-processing locale and the server when the following conditions are true:

- The **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables are set such that the code sets of the server-processing locale and the server locale are different.
- A valid code-set conversion exists between the code sets of the server locale and server-processing locale.

For a list of files for which Informix database servers perform code-set conversion, see [“Using the Server Locale” on page 4-3](#) for the Universal Server database server, [page 5-3](#) for the OnLine database server, and [page 6-3](#) for the SE database server. For information on GLS code-set conversion files, see [page A-14](#).

Which Data Is Converted?

When the code sets of two locales differ, an Informix product must use code-set conversion to prevent data corruption of character data. Code-set conversion converts the following types of character data:

- SQL data types
 - CHAR and VARCHAR
 - NCHAR and NVARCHAR
 - TEXT (the BYTE data type is *not* converted)
- Any of the ESQL/C character data types (**char**, **fixchar**, **string**, and **varchar**)
- SQL statements, both static and dynamic
- Database objects such as:
 - Column names
 - Table names
 - Statement-identifier names
 - Cursor names
- Stored procedure text
- Command text
- Error message text in the **sqlca.sqlerrm** field



Tip: If your ESQL/C client application uses code-set conversion, you might need to take special programming steps. For more information, see [“Handling Code-Set Conversion” on page 8-12](#).

UNIX

Locating Message Files

Informix products use GLS locales to locate product-specific message files. By default, Informix products automatically search a subdirectory that is associated with the client locale for the product-specific message files.

Informix products search the `$INFORMIXDIR/msg/lg_tr/code_set` directory. ♦

Windows

Informix products search the `%INFORMIXDIR%\msg\lg_tr\code_set` directory. ♦

In this path, *lg* and *tr* are the language and territory, respectively, from the name of the client locale, and *code_set* is the condensed form of the code-set name. For more information about condensed code-set names, see [“Locale-File Subdirectories” on page A-10](#).

Informix products use a precedence of environment variables to locate product-specific message files. The **DBLANG** environment variable lets you override the client locale, as the location of message files that Informix products use. You might use **DBLANG** to specify a directory where the message files reside for each locale that your environment supports. For more information, see the entry for **DBLANG** in [Chapter 2, “GLS Environment Variables.”](#)

How Do You Customize Client End-User Formats?

You can set environment variables to override the following end-user formats in the client locale:

- End-user format of date and time (DATE, DATETIME) values
- End-user format of monetary (MONEY) values

This section explains how to customize these end-user formats. For a description of an end-user format, see [page 1-12](#).

Customizing Date and Time End-User Formats

The GLS locales define end-user formats for dates and times, which you do not usually need to change. However, you can customize end-user formats for DATE and DATETIME values (for example, 10-27-95 for the date 10/27/95) with the following environment variables.

Environment Variable	Description
GL_DATE	Supports extended format strings for international formats in date end-user formats.
GL_DATETIME	Supports extended format strings for international formats in time end-user formats.
DBDATE	Specifies a date end-user format. <i>(Supported for backward compatibility.)</i>
DBTIME	Specifies a time end-user format for certain embedded-language (ESQL) library functions. <i>(Supported for backward compatibility.)</i>

A date or time end-user format string specifies a format for the manipulation of internal DATE or DATETIME values as a literal string. For more information, see [“End-User Formats” on page 1-12](#).

Tip: When you set these environment variables, you do not affect the internal format of the DATE and DATETIME values within a database.

The GL_DATE and GL_DATETIME environment variables support formatting directives that allow you to specify an end-user format. A formatting directive has the form %x (where x is one or more conversion characters). For the complete syntax of the GL_DATE and GL_DATETIME environment variables, see the entries for these environment variables in [“GLS-Related Environment Variables” on page 2-4](#).



Era-Based Date and Time Formats

The `GL_DATE` and `GL_DATETIME` environment variables provide support for alternative dates and times such as era-based (Asian) formats. These alternative formats support dates such as the Taiwanese Ming Guo year and the Japanese Imperial-era dates. For more information about these era-based date formats, see [“Date and Time Formats” on page 1-15](#).



Tip: The `DBDATE` and `DBTIME` environment variables also provide some support for era-based dates. See the entries for these environment variables in [Chapter 2, “GLS Environment Variables.”](#)

To specify era-based formats for `DATE` and `DATETIME` values, use the `E` conversion modifier, as follows:

- For either `GL_DATE` or `GL_DATETIME`, the `E` can appear in several formatting directives.
For a list of valid formatting conversions for eras, see [“Alternative Time Formats” on page 2-37](#).
- For `DBDATE`, the `E` can appear in the format specification.
For more information about date specifications, see the description of `DBDATE` in [Chapter 2, “GLS Environment Variables.”](#)

Date and Time Precedence

Informix products use the following precedence to determine the end-user format for an internal `DATE` value:

1. `DBDATE`
2. `GL_DATE`
3. Information that the client locale defines (`CLIENT_LOCALE`, if it is set)
4. `LC_TIME` (if `DBNLS` is set)

5. **LANG** (if **DBNLS** is set, and **LC_TIME** is not set)
If **LANG** is set, but **LC_TIME** is not, the client application sets **LC_TIME** from the **LANG** value; it does not send **LANG** to the database server. ♦
6. Default date format = %m/%d/%iy (if **DBDATE**, **GL_DATE** and **DBNLS** are not set, and no locale is specified)
For more information on these formatting directives, see the description of **GL_DATE** on [page 2-25](#).

Informix products use the following precedence to determine the end-user format for an internal DATETIME value:

1. **DBDATE** and **DBTIME**
2. **GL_DATETIME**
3. Information that the client locale defines (**CLIENT_LOCALE**, if it is set)
4. **LC_TIME** (if **DBNLS** is set)
If **LANG** is set, but **LC_TIME** is not, the client application sets **LC_TIME** from the **LANG** value; it does not send **LANG** to the database server.
5. **LANG** (if **DBNLS** is set, but **LC_TIME** is not set) ♦
6. Default DATETIME format = %iY-%m-%d %H:%M:%S (if **CLIENT_LOCALE**, **DBTIME**, **GL_DATETIME**, and **DBNLS** are not set)
For more information on these formatting directives, see the description of **GL_DATE** and **GL_DATETIME** on [pages 2-25](#) and [2-35](#), respectively.

Customizing Monetary Values

The GLS locales contain end-user formats, which you do not usually need to change. However, you can set the **DBMONEY** environment variable to customize the appearance of the currency notation. For information on the **DBMONEY** environment variable, see Chapter 4 of the [Informix Guide to SQL: Reference](#).

A monetary end-user format string specifies a format for the manipulation of internal DECIMAL, FLOAT, and MONEY values as monetary literal strings. For more information, see “[End-User Formats](#)” on page 1-12. Informix products use the following precedence to determine the end-user format for a MONEY value:

1. **DBMONEY**
 2. Information that the client locale defines (**CLIENT_LOCALE**, identifies the client locale; if it and **DBNLS** are not set, then the client locale is the default locale)
 3. **LC_MONETARY** (if **DBNLS** is set)
 4. **LANG** (if **DBNLS** is set, but **LC_MONETARY** is not set)
If **LANG** is set, but **LC_MONETARY** is not, the client application sets **LC_MONETARY** from the **LANG** value; it does not send **LANG** to the database server. ♦
 5. Default currency notation = \$,.
- If **DBMONEY** and **DBNLS** are not set, and no locale is specified, the currency symbol is the dollar sign, the thousands separator is a comma, and the decimal separator is the period.

GLS Environment Variables

GLS-Related Environment Variables	2-4
CC8BITLEVEL	2-5
CLIENT_LOCALE	2-6
DBDATE	2-7
DBLANG	2-15
DB_LOCALE	2-17
DBMONEY	2-19
DBTIME	2-20
ESQLMF	2-22
GLS8BITFSYS	2-23
GL_DATE	2-25
GL_DATETIME	2-35
SERVER_LOCALE	2-41

UNIX

Windows

Informix products establish the client, database, and server locales with information from GLS-related environment variables and from data that is stored in the database. This chapter provides descriptions of the GLS-related environment variables.

On UNIX platforms, you set environment variables with the appropriate shell command (such as **setenv** for the C shell). For more information, refer to your UNIX shell documentation. ♦

In Windows environments, you set environment variables in the Registry (with the SETNET32 utility) or in the **InetLogin** structure. For more information on SETNET32, see the *Installation and Configuration Guide for Microsoft Windows Environments*. For more information on **InetLogin**, see the Microsoft Windows supplement to your SQL API documentation. ♦

GLS-Related Environment Variables

Figure 2-1 shows an alphabetical list of the GLS-related environment variables that you can set for Informix database servers and SQL API products. These environment variables are described in this chapter on the pages that are listed in the last column set.

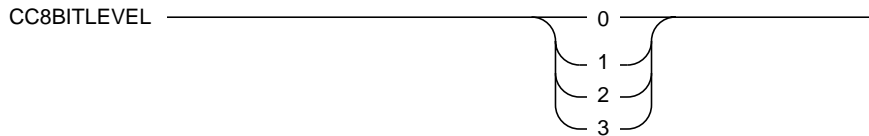
Environment Variable	Restrictions	Page
CC8BITLEVEL	ESQL/C only	2-5
CLIENT_LOCALE		2-6
DBDATE		2-7
DBLANG		2-15
DB_LOCALE		2-17
DBMONEY		2-19
DBTIME	SQL APIs only	2-20
ESQLMF	ESQL/C only	2-22
GLS8BITFSYS		2-23
GL_DATE		2-25
GL_DATETIME		2-35
SERVER_LOCALE		2-41

Figure 2-1
GLS-Related Environment Variables

Informix products support many other non-GLS environment variables. For a complete description of these environment variables, refer to Chapter 4 of the *Informix Guide to SQL: Reference*.

CC8BITLEVEL

The **CC8BITLEVEL** environment variable indicates the ability of your C-language compiler to process non-ASCII (8-bit and multibyte) characters.



The value of the **CC8BITLEVEL** environment variable determines the type of processing that the **ESQL/C** filter, **esqlmf**, performs on non-ASCII characters. The following values are possible:

- 0 tells the **esqlmf** filter to convert all non-ASCII characters, in literal strings and comments to octal constants (for C compilers that do not support these uses of non-ASCII characters).
- 1 tells the **esqlmf** filter to convert non-ASCII characters in literal strings to octal constants but allow them in comments (some C compilers do support non-ASCII characters in comments).
- 2 tells the **esqlmf** filter to allow non-ASCII characters in literal strings and ensure that all the bytes in the non-ASCII characters have the eighth bit set (for C compilers with this requirement).
- 3 indicates that the **esqlmf** filter does not need to filter non-ASCII characters (for C compilers that support multibyte characters in literal strings and comments).

To invoke **esqlmf** each time that you process an **ESQL/C** file with the **esql** command, set the **ESQLMF** environment variable to 1 (see [page 2-22](#)). If you do not set **CC8BITLEVEL**, the **esql** command assumes a value for **CC8BITLEVEL** of 0.



Important: For **ESQLMF** to take effect, do not set **CC8BITLEVEL** to 3.

For more information on the actions that **esqlmf** takes, see “[The esqlmf Filter](#)” on [page 8-7](#).

CLIENT_LOCALE

The **CLIENT_LOCALE** environment variable specifies the *client locale*, which the client application uses to perform read and write operations on the client computer. (For information about the client locale, see [page 1-23](#).)

CLIENT_LOCALE

Locale Name
p. 1-20

Informix products use the **CLIENT_LOCALE** environment variable for the following purposes:

- When the preprocessor for ESQL/C or ESQL/COBOL processes a source file, it accepts C or COBOL source code that is written in the code set of the **CLIENT_LOCALE**.

The C compiler and the operating system that you use might impose limitations on the ESQL/C program. For more information, see [“The esqlmf Filter” on page 8-7](#).

- When an ESQL/C or ESQL/COBOL client application executes, it checks **CLIENT_LOCALE** for the name of the client locale, which affects operating-system filenames, contents of text files, and formats of date, time, and numeric data.

For more information, see [“Non-ASCII Characters in ESQL Source Files” on page 7-3](#).

- When a client application and a database server exchange character data, the client application performs code-set conversion when the code set of the **CLIENT_LOCALE** environment variable is different from the code set of **DB_LOCALE** (on the client computer).

Code-set conversion prevents data corruption when these two code sets are different. For more information, see [“Performing Code-Set Conversion” on page 1-40](#).

- When the client application requests a connection, it sends information, including the **CLIENT_LOCALE**, to the database server.

The database server uses **CLIENT_LOCALE** when it determines how to set the client-application information of the server-processing locale. For more information, see [“Establishing a Database Connection” on page 1-31](#).

- When database utilities create files, the filenames and file contents are in the code set that **CLIENT_LOCALE** specifies.

If the operating system allows only ASCII characters in filenames, set the **GLS8BITFSYS** environment variable to tell database utilities that the operating system on the client computer is not 8-bit clean. For more information, see the description of **GLS8BITFSYS** on [page 2-23](#).

- When a client application looks for product-specific message files, it checks the message directory that is associated with the name of the client locale (**CLIENT_LOCALE**).

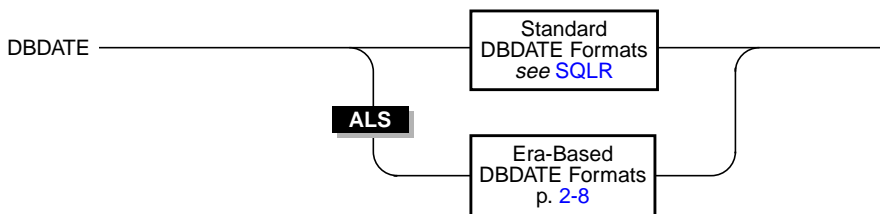
For more information, see [“Locating Message Files” on page 1-49](#).

If you do not set **CLIENT_LOCALE** (and **DBNLS** is also *not* set), the client application uses the default locale, U.S. English, as the client locale.

DBDATE

The **DBDATE** environment variable specifies the end-user formats of values in DATE columns. For more information on end-user formats, see [page 1-12](#).

Important: *DBDATE is evaluated at system initialization time. If it is invalid, the system initialization fails.*



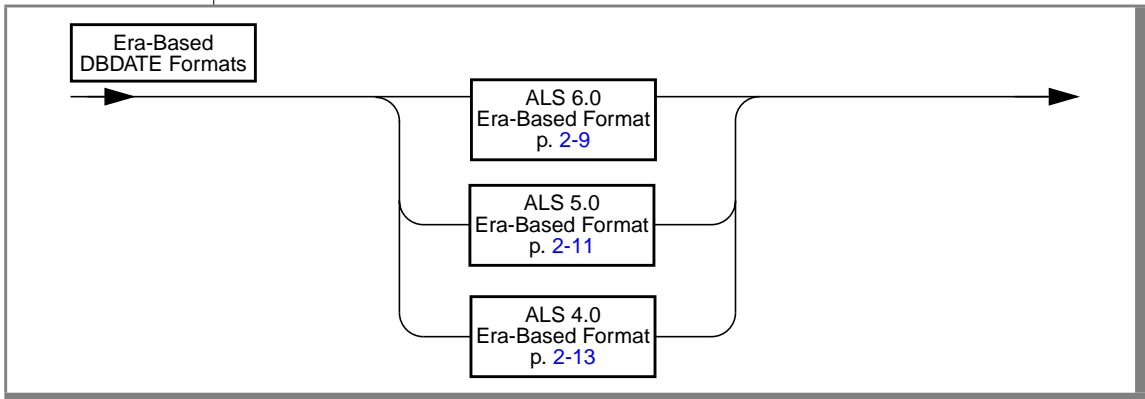
ALS

Informix products support the era-based extensions in **DBDATE** for backward compatibility with Informix ALS products. Informix recommends that you use the **GL_DATE** environment variable (see [page 2-25](#)) for date end-user formats in new client applications. ♦

For a description of the standard **DBDATE** formats and general information about this environment variable, refer to Chapter 4 of the [Informix Guide to SQL: Reference](#). This section describes the era-based (Asian) formats that **DBDATE** supports. For more information on era-based dates, see “[Era-Based Date and Time Formats](#)” on page 1-51.

Era-Based DBDATE Formats

The **DBDATE** environment variable supports special extensions that format values in DATE columns as era-based date strings. The following diagram shows the syntax of the era-based date **DBDATE** formats.



For **DBDATE** to format era-based dates, you must have a client locale that supports eras. The **CLIENT_LOCALE** environment variable must specify the name of a locale that defines era information. If the client locale does not support eras, Informix products ignore these era-based formats.



Tip: Era-based date formats can also appear in the date-format strings of certain *ESQL* library functions. For more information, see “[Extended DATE-Format Strings](#)” on page 7-10.

The following table shows some sample era-based **DBDATE** formats that are compatible with Informix ALS Version 6.0 products. In this table, the sample outputs for these formats assume that the client locale is Japanese SJIS (**ja_jp.sjis**).

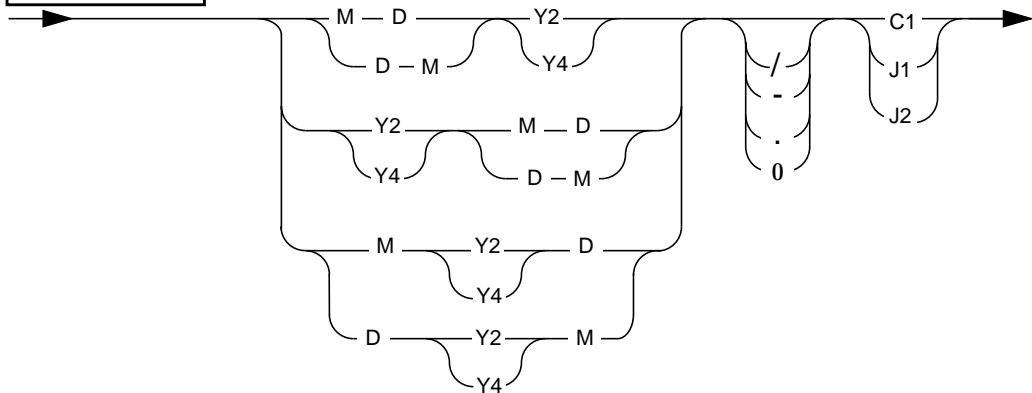
Description	DBDATE Setting	October 5, 1990 appears as:
Format with abbreviated era name	GY2MD/	H02/10/05
	MDGY2.	10.05.H02
	GY2MD0	H021005
	GY4DM/	H0002/05/10
Format with full era name	EY2MD/	A ¹ A ² 02/10/05
	MDEY20	1002A ¹ A ² 02
	EY2DM.	A ¹ A ² 02.05.10
	EY4MD/	A ¹ A ² 0002/10/05
	EY4DM-	A ¹ A ² 0002-05-10

In the preceding table, A¹A² is a multibyte character in the Japanese SJIS code set that represents the full era name. The H is an ASCII character that represents the abbreviated era name. The Japanese SJIS locale (**ja_jp.sjis**) defines these era names.

ALS 5.0 Era-Based Format

The ALS 5.0 Era-Based Format syntax for **DBDATE** provides backward compatibility with Informix ALS Version 5.0 products.

ALS 5.0 Era-Based
Format



The C1, J1, and J2 era-year modifiers appear as an extension, at the end of the **DBDATE** format. They indicate how to format an era, as follows:

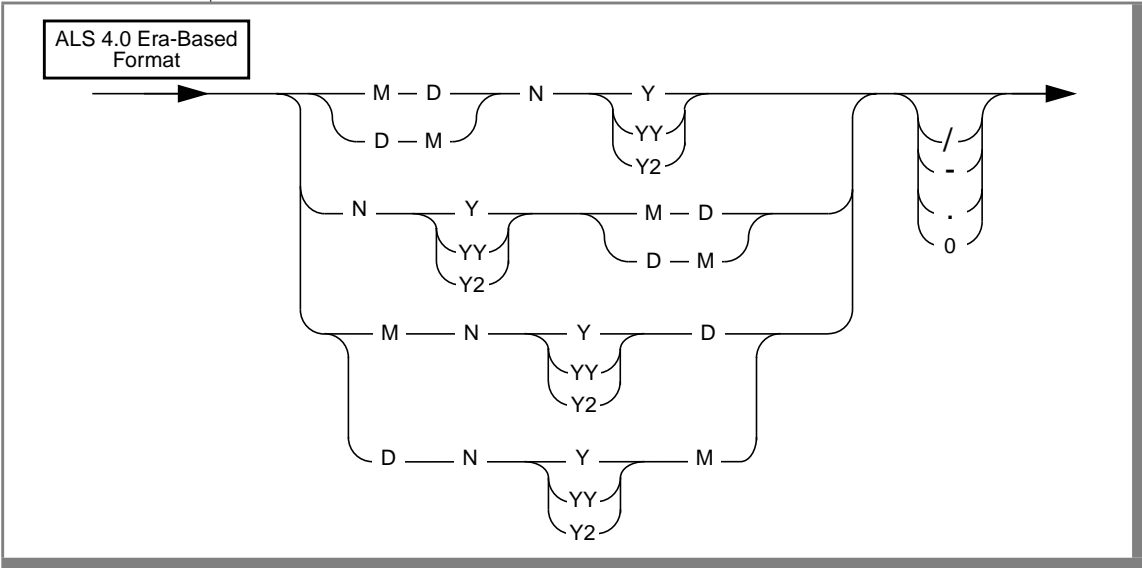
- The C1 era-year modifier formats the era as the abbreviated era name for a Taiwanese Ming Guo date.
- The J1 era-year modifier formats the era as the abbreviated era name for the Japanese Imperial era dates.
- The J2 era-year modifier formats the era as the full era name for the Japanese Imperial era dates.

The following table shows some sample era-based **DBDATE** formats that are compatible with Informix ALS Version 5.0 products. In this table, the sample outputs for the Japanese Imperial era assume that the locale is **ja_jp.sjis** (Japanese SJIS). The Taiwanese Ming Guo dates assume a **zh_tw.big5** locale.

Description	DBDATE Setting	October 5, 1990 appears as:
Japanese Imperial era	Y2MD-J1	H02-10-05
	Y4MD/J1	H0002/10/05
	Y2MD/J2	A ¹ A ² 02/10/05
	Y4MD/J2	A ¹ A ² 0002/10/05
	DMY4J2	05/10/A ¹ A ² 0002
Taiwanese Ming Guo date	Y2MD/C1	79/10/05
	Y4MD/C1	0079/10/05
	DMY4.C1	05.10.0079

ALS 4.0 Era-Based Format

The ALS 4.0 Era-Based Format syntax for **DBDATE** provides backward compatibility with Ascii Corporation Version 4.0 Asian products.



The following table shows some sample era-based **DBDATE** formats that are compatible with Ascii Corporation 4.0 products. In this table, the sample outputs for these formats assume that the locale is Japanese (**ja_jp.sjis**).

Description	DBDATE Setting	October 5, 1990 appears as:
Japanese Imperial era	NYMD/	H02/10/05
	NYYMD/	H02/10/05
	MDNYY/	10/05/H02
	NY2MD/	H02/10/05
	NY2DM-	H02-05-10

◆

Scanning and Printing Era-Based Formats

For an Informix product to *scan* a date string that contains an era-based date, it expects the **DBDATE** environment variable to specify the era-based format of the value to scan. Suppose that you set **DBDATE** to the following end-user format:

```
EY2DM/
```

An Informix product uses this era-based **DBDATE** format to interpret a date string that it must convert to an internal DATE value as follows:

- The **E** tells the Informix product to expect a multibyte character that represents the era name. For the Taiwanese locale (**zh_tw.big5**), you can omit the era name from the date string (the multibyte character is null).
- The **Y2** tells the Informix product to expect a year offset for the era. This offset can be a 2-digit or a 4-digit numeric value.
- The **D** tells the Informix product to expect a numeric value that represents the day of the month.
- The **M** tells the Informix product to expect either a numeric value that represents the month or the month name, which the locale defines.
- The slash (/) character identifies the date separator. However, the Informix product ignores the date separator when it scans a date string. Therefore, you can use any character as a date separator during the scan operation.

For an Informix product to *print* a date string that contains an era-based date, it expects **DBDATE** to indicate the era-based format of the value to print. With **DBDATE** set as in the previous example, an Informix product uses this end-user format to convert an internal DATE value to an era-based date string as follows:

- The **E** tells the Informix product to print the multibyte character that represents the era name.
- The **Y2** tells the Informix product to print a 2-digit year offset for the era.
- The **D** tells the Informix product to print the numeric value that represents the day.



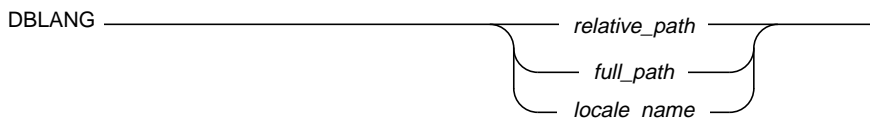
- The `M` tells the Informix product to print the numeric value that represents the month.
- The slash (/) character identifies the date separator that the Informix product prints between each portion of the date. This character must be one of the valid date separators during the print operation.

Tip: The `DBDATE` environment variable supports only the slash (/), period (.), and minus (-) as date separators. To specify some other character, including non-ASCII characters, as a date separator, use the `GL_DATE` environment variable.

For more information on the scan and print operations, see “[End-User Formats](#)” on page 1-12.

DBLANG

The `DBLANG` environment variable specifies the subdirectory of `INFORMIXDIR` that contains the customized, language-specific message files that an Informix product uses.



relative_path is the subdirectory of the Informix installation directory (that `INFORMIXDIR` specifies).

full_path is the full pathname of the directory that contains the compiled message files.

locale_name is the name of a GLS locale that has the format `lg_tr.code_set`, where `lg` is a two-character name that represents the language for a specific locale, `tr` is a two-character name that represents the cultural conventions, and `code_set` is the name of the code set that the locale supports.



Tip: For a detailed description of the syntax of `DBLANG`, see the “[Informix Guide to SQL: Reference](#).”

Informix products locate product-specific message files in the following order:

1. If **DBLANG** is set to a *full_path*: the directory that *full_name* indicates
2. If **DBLANG** is set to a *relative_path*:
 - a. In `$INFORMIXDIR/msg/$DBLANG`
 - b. In `$INFORMIXDIR/$DBLANG`
3. If **DBLANG** is set to a *locale_name*: the **msg** subdirectory for the locale in `$INFORMIXDIR/msg/lg_tr/code_set`, where *lg*, *tr*, and *code_set* are the language, territory, and code set, respectively, in *locale_name*.
 The value of **DBLANG** does not affect the messages that the database server writes to its message log. The database server obtains the locale for these messages from the **SERVER_LOCALE** environment variable.
4. If **DBLANG** is *not* set: the **msg** subdirectory for the locale in `$INFORMIXDIR/msg/lg_tr/code_set`, where *lg* and *tr* are the language and territory, respectively, from the locale that is associated with the Informix product, and *code_set* is the condensed name of the code set that the locale supports:
 - ❑ For Informix client products: *lg* and *tr* are from the client locale (from **CLIENT_LOCALE**, if it is set)
 - ❑ For Informix database server products: *lg* and *tr* are from the server locale (from **SERVER_LOCALE**, if it is set)
5. If **LANG** and **DBNLS** are set (the client uses NLS), but **DBLANG** is not set:
 - a. In `$INFORMIXDIR/msg/$LANG`
 - b. In `$INFORMIXDIR/$LANG` ♦

ALS

ALS

Windows

- 6. If **DBLANG**, **CLIENT_LOCALE**, and **LANG** are not set:
 - a. In **\$INFORMIXDIR/msg/en_us/0333**, an internal message directory for the default locale
 - b. In **\$INFORMIXDIR/msg/en_us.8859-1**, the message directory of the default locale (for backward compatibility with Version 6.0 ALS products) ♦
 - c. In **\$INFORMIXDIR/msg**, the default Informix message directories
 - d. In **\$INFORMIXDIR/msg/english**, the message directory for the default locale of the operating system ♦

This precedence specifies pathnames in UNIX syntax. In Windows environments, replace the syntax for the evaluation of an environment variable (**\$INFORMIXDIR**) with the corresponding Windows syntax (**%INFORMIXDIR%**). Also replace the slash (/) between directories with the backslash (\). For example, the UNIX **\$INFORMIXDIR/msg/\$DBLANG** pathname would be **%INFORMIXDIR%\msg\%DBLANG%** in a Windows environment. ♦

The compiled message files have the **.iem** file extension. For more information on **DBLANG**, see Chapter 4 of the [Informix Guide to SQL: Reference](#).

DB_LOCALE

The **DB_LOCALE** environment variable specifies the *database locale*, which the database server uses to handle locale-sensitive data types (NCHAR, NVARCHAR) of the database. (For information about the database locale, see [“The Database Locale” on page 1-26.](#))

DB_LOCALE

Locale Name
p. 1-20

Informix products use the **DB_LOCALE** environment variable for the following purposes:

- When a client application and a database server exchange character data, the client application performs code-set conversion when the value of the **DB_LOCALE** environment variable (on the client computer) is different from the value of **CLIENT_LOCALE**.

Code-set conversion prevents data corruption when these two code sets are different. For more information, see [“Performing Code-Set Conversion” on page 1-40](#).

- When the client application requests a connection, it sends information, including the **DB_LOCALE** (if it is set), to the database server.

The database server uses **DB_LOCALE** when it determines how to set the database information of the server-processing locale. For more information, see [“Sending the Client Locale” on page 1-32](#).

- When a client application tries to open a database, the database server compares the value of the **DB_LOCALE** environment variable that the client application passes with the database locale that is stored in the database.

When a database server accesses data in columns with locale-specific data types (NCHAR, NVARCHAR), it uses the locale that is saved in the database. For more information, see [“Verifying the Database Locale” on page 1-33](#).

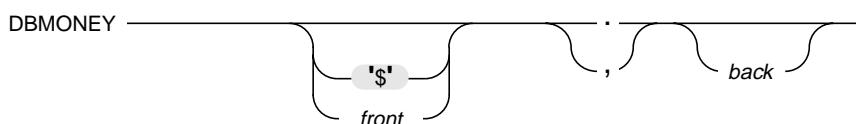
- When the database server creates a database, it examines the database locale (**DB_LOCALE**) to determine how to store character information in the system catalog of the database. If the database locale defines only a code-set order for collation (like the default locale, U.S. English), the database server creates CHAR and VARCHAR columns to store the character information. However, if the database locale defines a localized order for collation, the database server creates NCHAR and NVARCHAR columns to store this character information.

The database server uses the value of the **DB_LOCALE** environment variable that the client application sends. However, if you do not set **DB_LOCALE** on the client computer, the database server uses the value of **DB_LOCALE** on the server computer as the database locale.

For client applications, if you do not set **DB_LOCALE** on the client computer, the client applications assume that the database locale is the value of the **CLIENT_LOCALE** environment variable. However, the client application does not send this assumed value to the database server when it requests a connection.

DBMONEY

The **DBMONEY** environment variable specifies the end-user formats for values in MONEY columns. For more information about end-user formats, see [page 1-12](#).



Tip: For a detailed description of the syntax for **DBMONEY**, see Chapter 4 of the *“Informix Guide to SQL: Reference.”*

With this environment variable, you can set the following currency notation:

- The currency symbol that appears before or after the monetary value
The *front* value specifies a currency symbol before the monetary value and the *back* value specifies a currency symbol after the value. The currency symbol can be non-ASCII character(s) if your client locale supports a code set that defines the non-ASCII character(s) that you use.
- The monetary decimal separator, which separates the integral part of the monetary value from the fractional part
You can specify a comma (,) or a period (.) as the monetary decimal separator. When you use one of these symbols in the **DBMONEY** value, you implicitly assign the other symbol to the thousands separator.

For example, suppose you set **DBMONEY** as follows:

DM,

This value sets the following currency notation:

- The currency symbol, DM, appears before a monetary value.
- The decimal separator is a comma.
- The thousands separator is a period.

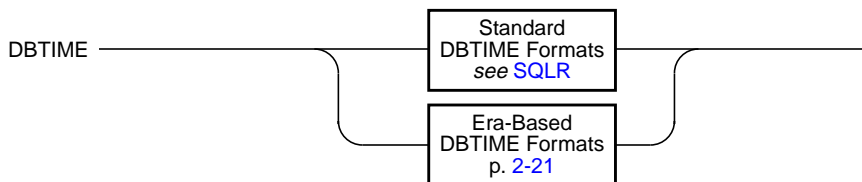
In the default locale, the currency symbol is the dollar sign (\$), and it appears at the front of the monetary values. The period (.) is the decimal separator, and the comma (,) is the thousands separator. The currency notation that you specify with **DBMONEY** takes precedence over the currency notation that the locale defines. For more information, see [“Customizing Monetary Values” on page 1-53](#).

DBTIME

The **DBTIME** environment variable specifies the end-user formats of values in DATETIME columns for SQL API routines. For more information on end-user formats, see [page 1-12](#).



***Tip:** The **DBTIME** environment variable affects only certain DATETIME and INTERVAL formatting routines in the INFORMIX-ESQL/COBOL and INFORMIX-ESQL/C function libraries. For information about how these library functions are affected, refer to [“DATETIME-Format Functions” on page 7-15](#).*



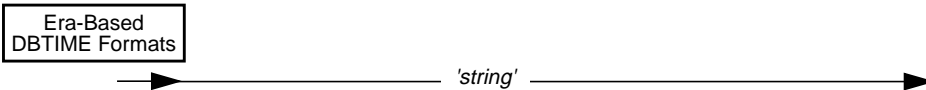
ALS

Informix products support the era-based extension in **DBTIME** for backward compatibility with Informix ALS products. Informix recommends that you use the **GL_DATETIME** environment variable (see [page 2-35](#)) for date and time end-user formats in Version 9.1 client applications. ♦

For a description of the standard **DBTIME** formats and general information about this environment variable, refer to the [Informix Guide to SQL: Reference](#). This section describes the era-based (Asian) formats that **DBTIME** supports. For more information on era-based dates and times, see “[Era-Based Date and Time Formats](#)” on page 1-51.

Era-Based DBTIME Formats

The **DBTIME** environment variable supports special extensions to format values in DATETIME columns as era-based dates and times. The following diagram shows the syntax for the era-based formatting directives that **DBTIME** supports.



string The formatting directives that specify the end-user format for the DATETIME values. You can use any formatting directive that formats non-era-based dates and times. For a list of these formatting directives, see the entry for **DBTIME** in Chapter 4 of the [Informix Guide to SQL: Reference](#).

You can also use the era-based formatting directives that are described in the following list:

- %EC** accepts either the full or the abbreviated era name during a scan; during a print, **%EC** is replaced by the full name of the base year of the era that the locale defines (same as **%C** if locale does not define an era).
- %Eg** accepts either the full or the abbreviated era name during a scan; during a print, **%Eg** is replaced by the abbreviated name of the base year of the era that the locale defines (same as **%C** if locale does not define an era).
- %Ey** is replaced by the offset from the start of the era that **%EC** or **%Eg** specifies. This date is the era year only (same as **%y** if locale does not define an era).

Because DATETIME values must be compliant with ANSI-SQL standards, you cannot specify a literal DATETIME string that does not comply with these standards in an SQL statement.

The following examples show valid DATETIME input values for the given DBTIME formats.

DBTIME Format	December 27, 1991 appears as:
%b %d, %Y	Dec 27, 1991
%y %m %dc1	80 12 27
%Y %m %dc1	0080 12 27
%Eg%Ey %m %d %H:%M:%S	H03 12 27 22:12:10

You can include non-ASCII characters in an era-based DBTIME format if the client locale supports a code set that contains these characters. For example, suppose you set CLIENT_LOCALE to the Japanese SJIS locale (**ja_jp.sjis**). To specify the multibyte SJIS character A¹A² as the separator between hours, minutes, and seconds, you might set DBTIME to the following string:

```
%EC%Ey %m %d %HA1A2%MA1A2%SA1A2
```

With this DBTIME value and a Japanese client locale of **ja_jp.sjis**, a DATETIME value of “December 27, 1991 22:12:10” would be formatted as follows:

```
B1B203 12 27 22A1A212A1A222A1A2
```

In the preceding example, the multibyte character B¹B² is the full era name, which the client locale must define.

ESQLMF

The ESQLMF environment variable indicates whether the **esql** command automatically invokes the ESQL/C multibyte filter, **esqlmf**.





When you set **ESQLMF** to 1, the **esql** command invokes **esqlmf** to filter multibyte characters as part of the preprocessing for an ESQL/C source file. The value of the **CC8BITLEVEL** environment variable (see [page 2-5](#)) determines the type of filtering that **esqlmf** performs. For more information on the actions that **esqlmf** takes, see [“The esqlmf Filter” on page 8-7](#).

Important: For **ESQLMF** to take effect, **CC8BITLEVEL** must not be set to 3.

If you want to compile existing source code whose non-ASCII characters have already been converted, either set **ESQLMF** to 0 or do not set it. In either case, **esql** does not invoke **esqlmf**.

GLS8BITFSYS

The **GLS8BITFSYS** environment variable tells an Informix product whether the operating-system can support non-ASCII characters in filenames.



When you set **GLS8BITFSYS** to 1 (or do not set it), Informix products can use non-ASCII characters (8-bit or multibyte characters) in a filename of an operating-system file that it generates. When you set **GLS8BITFSYS** to 0, Informix products generate filenames with 7-bit ASCII characters only.

Restrictions on Non-ASCII Filenames

If your locale supports a code set with non-ASCII characters, restrictions apply to filenames for operating-system files that Informix products generate. Before you or an Informix product creates a file and assigns a filename, consider the following questions:

- Does your operating system support non-ASCII filenames?
- Does the Informix product accept non-ASCII filenames?

Making Sure That Your Operating System Is 8-Bit Clean

To support non-ASCII characters in filenames, your operating system must be *8-bit clean*. An operating system is 8-bit clean if it reads the eighth bit as part of the code value. In other words, the operating system must not ignore or make its own interpretation on the value of the eighth bit.

Informix recommends that you consult your operating-system manual or system administrator to determine whether your operating system is 8-bit clean before you decide to use a nondefault locale that contains non-ASCII characters in filenames that Informix products use and generate.

Setting the GLS8BITFSYS Environment Variable

Use the **GLS8BITFSYS** environment variable to tell Informix products whether the operating system is 8-bit clean. This environment variable determines whether an Informix product can use non-ASCII characters in a filename of an operating-system file that it generates:

- When you set **GLS8BITFSYS** to 1, Informix products assume that the operating system is 8-bit clean.

An Informix product can use non-ASCII characters in a filename. If you include non-ASCII characters in a filename that you specify from within a client application, you must ensure that the code set of the server-processing locale supports these non-ASCII characters. If you do not set **GLS8BITFSYS**, Informix database servers behave as if **GLS8BITFSYS** is set to 1.

- When you set **GLS8BITFSYS** to 0, Informix products assume that the operating system is *not* 8-bit clean.

An Informix product does *not* allow non-ASCII characters in filenames. If you include non-ASCII characters in a filename from within a client application, the Informix product uses an internal algorithm to convert these non-ASCII characters to ASCII characters. The filenames that result are 7-bit clean.

Filenames with invalid byte sequences generate errors when they are used with GLS-based products.

Operating-System Files That Informix Products Generate

The value of the **GLS8BITFSYS** environment variable affects the filenames of the following types of files that Informix products generate:

- INFORMIX-Universal Server and INFORMIX-OnLine Dynamic Server use **GLS8BITFSYS** on the server computer to create files for a message log and diagnostic information.

For a list of the Universal Server files, see [“Using the Server Locale” on page 4-3](#); for a list of OnLine files, see [“Using the Server Locale” on page 5-3](#).

- The INFORMIX-SE database server uses **GLS8BITFSYS** on the server computer to create files for database and table names.

For a list of these SE files, see [“Using the Server Locale” on page 6-3](#).

- Some database utilities, such as **dbexport**, use **GLS8BITFSYS** on the client computer to create files.

- The compilers for INFORMIX-ESQL/C and INFORMIX-ESQL/COBOL products use **GLS8BITFSYS** on the client computer to create and use ESQL source files.

For a list of these files, see [“Non-ASCII Characters in ESQL Source Files” on page 7-3](#).

Once an Informix product has generated an operating-system file, it has written that filename and the file contents in a particular code set. Whenever an Informix product or client application needs to access that file, you must ensure that the product uses a server-processing locale that supports that same code set. For more information, refer to the section for your Informix product that the preceding list names.

GL_DATE

The **GL_DATE** environment variable specifies end-user formats of values for DATE columns. For more information on end-user formats, see [page 1-12](#).

Important: The **GL_DATE** variable is evaluated when it is used. If it is invalid, the operations that called it will fail.

The extended functionality of the **GL_DATE** environment variable replaces the functionality of the **DBDATE** environment variable (see [page 2-7](#)). Informix recommends that applications use **GL_DATE** instead of **DBDATE**.



An end-user format in **GL_DATE** can contain the following characters:

- One or more white-space characters, which the CTYPE category of the locale specifies (For more information on the CTYPE locale category, see [page A-6](#).)
- An ordinary character (other than the % symbol or a white-space character)
- A formatting directive, which is composed of the % symbol followed by a conversion character that specifies the required replacement

GL_DATE _____ 'string' _____

string The formatting directives that specify the end-user format for **GL_DATE** values. You can use any formatting directive that formats dates. For a list of the era-based formatting directives, see “[Alternative Date Formats](#)” on [page 2-29](#).

You can also use the non-era-based formatting directives that are described in the following list:

- %a is replaced by the abbreviated weekday name as defined in the locale.
- %A is replaced by the full weekday name as defined in the locale.
- %b is replaced by the abbreviated month name as defined in the locale.
- %B is replaced by the full month name as defined in the locale.
- %C is replaced by the century number (the year divided by 100 and truncated to an integer) as a decimal number (00 through 99).
- %d is replaced by the day of the month as a decimal number (01 through 31). A single digit is preceded by a zero (0).
- %D is the same as the %m/%d/%y format.

- %e** is replaced by the day of the month as a decimal number (1 through 31). A single digit is preceded by a space.
- %h** is the same as the **%b** formatting directive.
- %iy** is replaced by the year as a 2-digit decade (00 through 99) for both scanning and printing. It is the Informix-specific formatting directive for **%y**. For more information, see [“The Year Formatting Directives” on page 2-28](#).
- %iY** is replaced by the year as a 4-digit decimal number (0000 through 9999) for both scanning and printing. It is the Informix-specific formatting directive for **%Y**. For more information, see [“The Year Formatting Directives” on page 2-28](#).
- %m** is replaced by the month as a decimal number (01 through 12).
- %n** is replaced by a new-line character.
- %t** is replaced by the TAB character.
- %w** is replaced by the weekday as a decimal number (0 through 6); 0 represents the locale equivalent of Sunday.
- %x** is replaced by a special date representation that the locale defines.
- %y** requires that the year be a 2-digit decimal number (00 through 99) for both scanning and printing. For more information, see [“The Year Formatting Directives” on page 2-28](#).
- %Y** requires that the year be a 4-digit decimal number (0000 through 9999) for both scanning and printing. For more information, see [“The Year Formatting Directives” on page 2-28](#).
- %%** is replaced by **%** (to allow **%** in the format string).

White-space or other nonalphanumeric characters must appear between any two formatting directives. For example, if you use a U.S. English locale, you might want to format an internal DATE value for 03/05/1996 to the ASCII string format that the following example shows:

```
Mar 05, 1996 (Tuesday)
```

To do so, set the **GL_DATE** environment variable as follows:

```
%b %d, %Y (%A)
```

If a **GL_DATE** format does not correspond to any of the valid formatting directives, the behavior of the Informix product when it tries to format is undefined.

***Tip:** The **GL_DATETIME** environment variable accepts these date-formatting directives in addition to those that the section on [page 2-35](#) lists.*



The Year Formatting Directives

You can use the following formatting directives in the end-user format of the **GL_DATE** environment variable to format the year of a date string: **%y**, **%iy**, **%Y**, and **%iY**. The **%iy** and **%iY** formatting directives provide compatibility with the Y2 and Y4 year specifiers of the **DBDATE** environment variable.

When an Informix product uses an end-user format to *print* an internal date value as a string, the **%iy** and **%iY** formatting directives perform the same task as **%y** and **%Y**, respectively. To print a year with one of these formatting directives, an Informix product performs the following actions:

- The **%iy** and **%y** formatting directives both print the year of an internal date value as a 2-digit decade.

For example, when you set **GL_DATE** to '**%y %m %d**' or '**%iy %m %d**', an internal date for March 6, 1996 formats to '96 03 06'.

- The **%iY** and **%Y** formatting directives both print the year of an internal date value as a 4-digit year.

For example, when you set **GL_DATE** to '**%Y %m %d**' or '**%iY %m %d**', the internal date for March 6, 1996 formats to '1996 03 06'.

When an Informix product uses an end-user format to *scan* a date, the %iy and %iY formatting directives perform differently from %y and %Y, respectively. The following table summarizes how the year formatting directives behave when an Informix product uses them to scan date strings.

GL_DATE Format	Date String to Scan:	
	'1994 03 06'	'94 03 06'
%y %m %d	<i>error</i>	internal date for 1994 03 06
%iy %m %d	internal date for 1994 03 06	internal date for 1994 03 06
%Y %m %d	internal date for 1994 03 06	<i>internal date for 0094 03 06</i>
%iY %m %d	internal date for 1994 03 06	internal date for 1994 03 06

In a scan of a date string, the %iy and %y formatting directives both assume the current century for a 2-digit year. However, you can set the **DBCENTURY** environment variable to change this assumption.

For more information on how Informix products use end-user formats to scan and print strings, see [“End-User Formats” on page 1-12](#).

Alternative Date Formats

To support alternative date formats in an end-user format, **GL_DATE** accepts the following *conversion modifiers*:

- E indicates use of an alternative era format, which the locale defines.
- 0 (the letter “O”) indicates use of alternative digits, which the locale also defines.

The following table shows date-formatting directives that support conversion modifiers.

Alternative Date Format	Description
%EC	accepts either the full or the abbreviated era name for scanning; for printing, %EC is replaced by the full name of the base year (period) of the era that the locale defines (same as %C if locale does not define an era).
%Eg	accepts either the full or the abbreviated era name for scanning; for printing, %Eg is replaced by the abbreviated name of the base year (period) of the era that the locale defines (same as %C if locale does not define an era).
%Ex	is replaced by a special date representation for an era that the locale defines (same as %x if locale does not define an era).
%Ey	is replaced by the offset from %EC of the era that the locale defines. This date is the era year only (same as %y if locale does not define an era).
%EY	is replaced by the full era year, which the locale defines (same as %Y if locale does not define an era).
%Od	is replaced by the day of the month in the alternative digits that the locale defines (same as %d if locale does not define alternative digits).
%Oe	is the same as %Od (same as %e if locale does not define alternative digits).
%Om	is replaced by the month in the alternative digits that the locale defines (same as %m if locale does not define alternative digits).
%Ow	is replaced by the weekday as a single-digit number (0 through 6) in the alternative digits that the locale defines (same as %w if locale does not define alternative digits). The equivalent of zero (0) represents the locale equivalent of Sunday.
%Oy	is replaced by the year as a 2-digit number (00 through 99) in the alternative digits that the locale defines (same as %y if locale does not define alternative digits). For information about how to format a year value, see the description of %y.
%OY	is the same as %EY (same as %Y if locale does not define alternative digits).

The TIME category of the locale defines the following era information:

- The full and abbreviated names for an era
- A special date representation for the era (which the %Ex formatting directive uses)

The NUMERIC category of the locale defines the alternative digits for a locale (which the %Ox formatting directives use). For more information on the TIME and NUMERIC categories of a locale, see [page A-4](#).

For a description of the era-based formats for DATETIME values, see “[Alternative Time Formats](#)” on page 2-37.

Optional Date Format Qualifiers

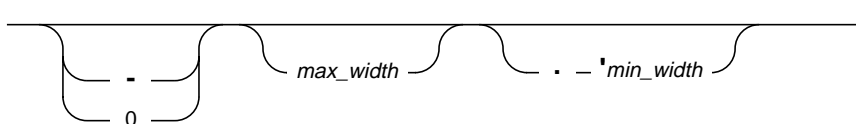
You can specify the following optional *format qualifiers* immediately after the % symbol of the formatting directive. A date format qualifier defines a field specification for the date that the Informix product scans or prints. The following sections describe what a field specification means for the scan and print operations. (For more information on the scan and print operations, see “[End-User Formats](#)” on page 1-12.)



Tip: The `GL_DATETIME` environment variable accepts these date format qualifiers in addition to those that “[Optional Time Format Qualifiers](#)” on page 2-38 lists.

Field Specification for a Scan

When an Informix product uses an end-user format to scan a date string, the field specification defines the number of characters to expect as input. This field specification has the following syntax.



max_width A decimal number that indicates the maximum number of characters to scan

min_width A decimal number that indicates the minimum number of characters to scan

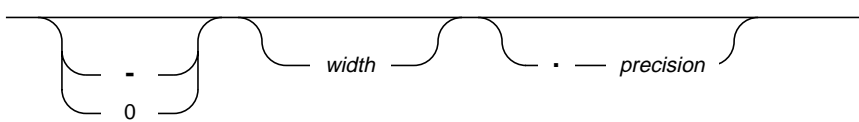
The first character of the field specification indicates whether to assume that the field value is justified or padded, as follows:

- If the first character is a minus sign (-), an Informix product assumes that the field value is *left justified* and begins with a digit; this value can include trailing spaces.
- If the first character is a zero (0), an Informix product assumes that the field value is *right justified* and any zeros on the left are pad characters (they are not significant).
- If the first character is neither a minus sign nor a zero (0), the Informix product assumes that the field value is *right justified* and any spaces on the left are pad characters. However, if the field value begins with a zero, it cannot include pad characters.

An Informix product ignores the field specification if the field value is not a numeric value.

Field Specification for a Print

When an Informix product uses an end-user format to print a date string, the field specification defines the number of characters to print as output. This field specification has the following syntax.



width A decimal number that indicates a minimum field width for the printed value

precision A decimal number that indicates the precision to use for the field value. The meaning of the precision value depends on the particular formatting directive with which it is used, as the following table shows.

Formatting Directives	Description
%C, %d, %e, %Ey, %iy, %iY, %m, %w, %y, %Y	Value of <i>precision</i> specifies the minimum number of digits to print. If a value supplies fewer digits than <i>precision</i> specifies, an Informix product pads the value with leading zeros. The %d, %Ey, %iy, %m, %w, and %y formatting directives have a default precision of 2. The %Y directive has no precision default; year 0001 would be formatted as 1 rather than as 0001.
%a, %A, %b, %B, %EC, %Eg, %h	Value of <i>precision</i> specifies the maximum number of characters to print. If a value supplies more characters than <i>precision</i> specifies, an Informix product truncates the value.

(1 of 2)

Formatting Directives	Description
%D	Values of <i>width</i> and <i>precision</i> affect each element of these formatting directives. For example, %6.4D prints as follows: %6.4m/%6.4d/%6.4y
%Ox	For formatting directives that include the O modifier (alternative digits), the value of <i>precision</i> is still the minimum number of digits to print. The <i>width</i> value defines the format width rather than the actual number of digits.
%Ex, %EY, %n, %t, %x, %%	The values of <i>width</i> and <i>precision</i> have no effect on these formatting directives.

(2 of 2)

For example, the following formatting directive displays the month as a decimal number with a maximum field width of 4:

```
% 4 m
```

The following formatting directive displays the day of the month as a decimal number with a minimum field width of 3 and a maximum field width of 4:

```
% 4 . 3 d
```

The first character of the field specification indicates whether to justify or pad the field value, as follows:

- If the first character is a minus sign (-), an Informix product prints the field value as *left justified* and pads this value with spaces on the right.
- If the first character is a zero (0), an Informix product prints the field value as *right justified* and pads this value with zeros on the left.
- If the first character is neither a minus sign nor a zero (0), an Informix product prints the field value as *right justified* and pads this value with spaces on the left.

GL_DATETIME

The `GL_DATETIME` environment variable specifies the end-user formats of values in DATETIME columns. For more information on end-user formats, see [page 1-12](#).



Important: The `GL_DATETIME` environment variable supports a superset of the formatting directives of the `DBTIME` environment variable (see [page 2-20](#)). Informix recommends that applications use `GL_DATETIME` instead of `DBTIME`.

A `GL_DATETIME` format can contain the following characters:

- One or more white-space characters, which the CTYPE category of the locale specifies (For more information on the CTYPE locale category, see [page A-6](#).)
- An ordinary character (other than the % symbol or a white-space character)
- A formatting directive, which is composed of the % symbol followed by a conversion character that specifies the required replacement

`GL_DATETIME` `'string'`

string The formatting directives that specify the end-user format for `GL_DATETIME` values. You can use any formatting directive that formats dates or times. For a list of formatting directives for dates, see the description of `GL_DATE` on [page 2-25](#). For a list of the era-based formatting directives for time values, see “[Alternative Time Formats](#)” on [page 2-37](#).

You can also use the non-era-based time formatting directives that are described in the following list:

- `%c` is replaced by a special date/time representation that the locale defines.
- `%Fn` is replaced by the value of the fraction of a second with precision that is specified by the integer *n*. The default value of *n* is 2; the range of *n* is $0 \leq n \leq 5$. This value overrides any width or precision that is given between the % and F character (see [page 2-38](#)).

- %H is replaced by the hour as a decimal number (00 through 23) (24-hour clock).
- %I is replaced by the hour as a decimal number (00 through 12) (12-hour clock).
- %M is replaced by the minute as a decimal number (00 through 59).
- %p is replaced by the A.M. or P.M. equivalent as defined in the locale.
- %r is replaced by the commonly used time representation for a 12-hour clock format (including the A.M. or P.M. equivalent) as defined in the locale.
- %R is replaced by the time in 24-hour notation (%H:%M).
- %S is replaced by the second as a decimal number (00 through 61). The second can be up to 61 instead of 59 to allow for the occasional leap second and double leap second.
- %T is replaced by the time in the %H:%M:%S format.
- %X is replaced by the commonly used time representation as defined in the locale.

White-space or other nonalphanumeric characters must appear between any two formatting directives. If a **GL_DATETIME** format does not correspond to any of the valid formatting directives, the behavior of the Informix product when it tries to format is undefined.

In addition to the formatting directives that are listed in the preceding table, you can include the following date-formatting directives in the end-user format of **GL_DATETIME**:

%a, %A, %b, %B, %C, %d, %D, %e, %h, %iy, %iY, %m, %n, %t, %w,
%x, %y, %Y, %%

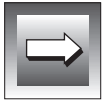
For more information on these date-formatting directives, see the **GL_DATE** environment variable on [page 2-25](#).

For example, if you use an U.S. English locale, you might want to format an internal DATETIME YEAR TO SECOND value to the ASCII string format that the following example shows:

```
Mar 21, 1995 at 16 h 30 m 28 s
```

To do so, set the **GL_DATETIME** environment variable as the following line shows:

```
%b %d, %Y at %H h %M m %S s
```



Important: The value of **GL_DATETIME** affects the behavior of certain ESQL library functions if the **DBTIME** environment variable ([page 2-20](#)) is not set. (For information about how these library functions are affected, refer to “[DATETIME-Format Functions](#)” on [page 7-15](#).) The value of **DBTIME** takes precedence over that of **GL_DATETIME**.

Alternative Time Formats

To support alternative time formats in an end-user format, **GL_DATETIME** accepts the following *conversion modifiers*:

- **E** indicates use of an alternative era format, which the locale defines.
- **O** (the letter “O”) indicates use of alternative digits, which the locale also defines.

The following table shows time-formatting directives that support conversion modifiers.

Alternative Time Format	Description
%Ec	is replaced by a special date/time representation for the era that the locale defines (same as %c if locale does not define an era).
%EX	is replaced by a special time representation for the era that the locale defines (same as %X if locale does not define an era).
%OH	is replaced by the hour in the alternative digits that the locale defines (24-hour clock) (same as %H if locale does not define alternative digits).

(1 of 2)

Alternative Time Format	Description
%OI	is replaced by the hour in the alternative digits that the locale defines (12-hour clock) (same as %I if locale does not define alternative digits).
%OM	is replaced by the minute with the alternative digits that the locale defines (same as %M if locale does not define alternative digits).
%OS	is replaced by the second with the alternative digits that the locale defines (same as %S if locale does not define alternative digits).

(2 of 2)

The TIME category of the locale defines the following era information:

- The full and abbreviated names for an era
- A special date representation for the era (which the %Ex formatting directive uses)
- A special time representation for the era (which the %EX formatting directive uses)
- A special date/time representation for the era (which the %Ec formatting directive uses)

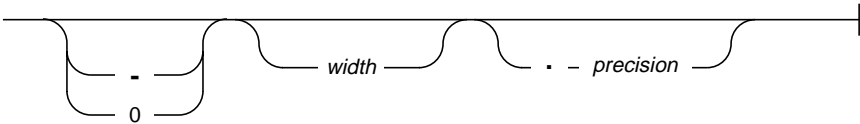
The NUMERIC category of the locale defines the alternative digits for a locale (which the %Ox formatting directives use). For more information on the TIME and NUMERIC categories of a locale, see [page A-4](#).

GL_DATETIME also supports the era-based date-formatting directives in an end-user format. For more information, see “[Alternative Date Formats](#)” on [page 2-29](#).

Optional Time Format Qualifiers

You can specify the following optional *format qualifiers* immediately after the % symbol of the formatting directive. A time format qualifier defines a field specification for the time (or date and time) that the Informix product scans or prints. This section describes what a field specification means for the print operation. For a description of what a field specification means for the scan operation, see [page 2-32](#). For general information on the scan and print operations, see “[End-User Formats](#)” on [page 1-12](#).

When an Informix product uses an end-user format to print a string from an internal format, the field specification defines the number of characters to print as output. This field specification has the following syntax.



- width* A decimal number that indicates a minimum field width for the printed value
- precision* A decimal number that indicates the precision to use for the field value. The meaning of the precision value depends on the particular formatting directive with which it is used, as the following table shows.

Formatting Directives	Description
%F, %H, %I, %M, %S	Value of <i>precision</i> specifies the minimum number of digits to print. If a value supplies fewer digits than the <i>precision</i> specifies, an Informix product pads the value with leading zeros. The %H, %M, and %S formatting directives have a default precision of 2.
%p	Value of <i>precision</i> specifies the maximum number of characters to print. If a value supplies more characters than the <i>precision</i> specifies, an Informix product truncates the value.
%R, %T	Values of <i>width</i> and <i>precision</i> affect each element of these formatting directives. For example, %6.4R prints as follows: %6.4H:%6.4M
%F	Value of <i>precision</i> can follow this directive as an optional precision specification. This value must be between 1 and 5. Otherwise, an Informix product generates an error. This precision value overrides any <i>precision</i> value that you specify between the % symbol and the formatting directive.

(1 of 2)

Formatting Directives	Description
%Ox	For formatting directives that include the O modifier, value of <i>precision</i> is still the minimum number of digits to print. The <i>width</i> value defines the format width rather than the actual number of digits.
%c, %Ec, %EX, %X	The values of <i>width</i> and <i>precision</i> have no effect on these formatting directives.

(2 of 2)

For example, the following formatting directive displays the minute as a decimal number with a maximum field width of 4:

%4M

The following formatting directive displays the hour as a decimal number with a minimum field width of 3 and a maximum field width of 6:

%6.3H

The first character of the field specification indicates whether to justify or pad the field value, as follows:

- If the first character is a minus sign (-), an Informix product prints the field value as *left justified* and pads this value with spaces on the right.
- If the first character is a zero (0), an Informix product prints the field value as *right justified* and pads this value with zeros on the left.
- If the first character is neither a minus sign nor a zero (0), an Informix product prints the field value as *right justified* and pads this value with spaces on the left.

SERVER_LOCALE

The **SERVER_LOCALE** environment variable specifies the *server locale*, which the database server uses to perform read and write operations that involve operating-system files on the server computer. (For information about the server locale, see [“The Server Locale” on page 1-27.](#))

SERVER_LOCALE

Locale Name p. 1-20

The database server uses the **SERVER_LOCALE** environment variable for the following purposes:

- When the Universal Server database server or the OnLine database server writes its operating-system files, it uses the server code set (that **SERVER_LOCALE** specifies).

For a list of Universal Server operating-system files, see [page 4-3](#); for a list of OnLine operating-system files, see [page 5-3](#).

- On operating systems that are 8-bit clean, the INFORMIX-SE database server stores the names of database objects (such as the database name and table names) in the code set that **SERVER_LOCALE** specifies.

On operating systems that are *not* 8-bit clean, SE can convert any non-ASCII characters in these names to ASCII characters. Set the **GLS8BITFSYS** environment variable to indicate whether the operating system is 8-bit clean. For more information, see [“Using the Server Locale” on page 6-3](#).

- When a database server looks for product-specific message files, it looks in the message directory that is associated with the name of the server locale (**SERVER_LOCALE**).

For more information, see [“Locating Message Files” on page 1-49](#).

For Informix database servers, if you do not set **SERVER_LOCALE**, these database servers use the default locale, U.S. English, as the server locale.

SQL Features

Naming Database Objects	3-4
Rules for Identifier Names	3-4
Identifiers That Support Non-ASCII Characters	3-5
References to SQL Segments	3-7
Owner Name in SQL Identifiers.	3-8
Valid Characters in Identifier Names	3-8
Non-ASCII Characters in Delimited Identifiers	3-10
Multibyte Characters and Identifier Length	3-10
Using Character Data Types.	3-11
The NCHAR Data Type	3-11
Collating NCHAR Data	3-12
Handling NCHAR Data	3-12
Multibyte Characters with NCHAR	3-12
Treating NCHAR Values as Numeric Values	3-13
Nonprintable Characters with NCHAR	3-13
The NVARCHAR Data Type	3-13
Collating NVARCHAR Data	3-14
Handling NVARCHAR Data.	3-15
Multibyte Characters with NVARCHAR.	3-15
Nonprintable Characters with NVARCHAR	3-15
Storing Numeric Values in an NVARCHAR Column	3-16
Performance Considerations for NCHAR and NVARCHAR	3-16
Other Character Data Types	3-17
The CHAR Data Type	3-17
The VARCHAR Data Type	3-18
The TEXT Data Type	3-19

Collating Character Data	3-20
Collation Order in CREATE INDEX	3-20
Collation Order in SELECT Statements.	3-21
The ORDER BY Clause	3-21
Logical Predicates in a WHERE Clause	3-24
Comparisons with MATCHES and LIKE Conditions.	3-28
Handling Character Data	3-33
Specifying Quoted Strings	3-33
Specifying Comments	3-34
Specifying Column Substrings.	3-34
Column Substrings in Single-Byte Code Sets	3-35
Column Substrings in Multibyte Code Sets	3-35
Partial Characters in Column Substrings	3-36
Specifying Arguments to the TRIM Function.	3-40
Using SQL Length Functions	3-40
The LENGTH Function	3-40
With Single-Byte Code Sets	3-41
With Multibyte Code Sets	3-42
The OCTET_LENGTH Function	3-43
The CHAR_LENGTH Function	3-45
Handling MONEY Columns.	3-47
Default Values for the Scale Parameter	3-48
Format of Currency Notation	3-49
Using Data Manipulation Statements.	3-49
Specifying Conditions in the WHERE Clause	3-50
Specifying Era-Based Dates.	3-50
Loading and Unloading Data	3-51
Loading Data into a Database	3-52
Unloading Data from a Database	3-52

This chapter explains how the GLS feature affects the Informix implementation of SQL. It describes how the choice of a locale affects the following topics:

- The names you use for database objects in data definition statements such as CREATE TABLE and CREATE INDEX
- The behavior of character data types such as CHAR, VARCHAR, NCHAR, and VARCHAR
- Collation order in the CREATE INDEX and SELECT statements
- The use of non-ASCII characters in quoted strings, comments, column substrings, and the TRIM function in SELECT and other statements
- The behavior of the MONEY data type
- The SQL length functions
- The behavior of data manipulation statements such as INSERT, DELETE, and UPDATE and load statements

For descriptions of SQL features that are not unique to the GLS feature, see the [Informix Guide to SQL: Syntax](#). For additional information on the Informix implementation of SQL, consult the [Informix Guide to SQL: Reference](#) and the [Informix Guide to SQL: Tutorial](#).

The information in this chapter pertains to all Informix database servers. INFORMIX-Universal Server extends the functionality of relational databases to object-oriented relational database with the creation of new data types. For information about how these new data types might affect global language support, see [Chapter 4, “INFORMIX-Universal Server Features.”](#)



Naming Database Objects

You need to assign names to database objects when you use data definition statements such as CREATE TABLE and CREATE INDEX. This section describes considerations for naming database objects when you use a nondefault locale. In particular, this section explains which SQL identifiers and delimited identifiers accept non-ASCII characters.

Important: To use a nondefault locale, you must set the appropriate locale environment variables for Informix products. For more information, see [“A Nondefault Locale” on page 1-28](#).

Rules for Identifier Names

An SQL identifier is a sequence of letters, digits, and underscores that represents the name of a database object such as a table, column, index, or view. The following table summarizes the rules for SQL identifiers.

SQL Identifier Rules	For More Information
An SQL identifier must begin with a letter or with an underscore. The remaining characters in the identifier can be any combination of letters, numbers, and underscores.	“Valid Characters in Identifier Names” on page 3-8
You cannot include white-space characters in identifiers unless you use them in a delimited identifier. You cannot use SQL reserved words as identifiers unless you use them in a delimited identifier.	“Non-ASCII Characters in Delimited Identifiers” on page 3-10
An SQL identifier can contain up to 18 bytes. The only exception is that database names are limited to 10 bytes in INFORMIX-SE.	“Multibyte Characters and Identifier Length” on page 3-10

Identifiers That Support Non-ASCII Characters

If you use a nondefault locale that supports a code set with non-ASCII characters, you can use these non-ASCII characters to form most SQL identifiers. Figure 3-1 shows the SQL identifiers that support non-ASCII characters for all Informix database servers prior to Universal Server. (See [Figure 4-3 on page 4-17](#) for additional supported SQL identifiers for Universal Server.) In this table, the SQL Identifier column lists the name of each database object. The SQL Segment column shows the segment that gives the complete syntax of the identifier in the *Informix Guide to SQL: Syntax*. (For more information, see [page 3-7](#).) The Notes column describes any special considerations for the identifier and also provides an example of an SQL statement that creates or uses the identifier.

All SQL identifiers in Figure 3-1 can be used with Universal Server databases, OnLine Dynamic Server databases, or INFORMIX-SE databases unless the Notes column indicates otherwise.

Figure 3-1
SQL Identifiers That Support Non-ASCII Characters

SQL Identifier	SQL Segment	Notes
Column name	Expression	An example of an SQL statement that creates a column name is CREATE TABLE.
Connection name	Quoted string	An example of an SQL statement that creates a connection name is CONNECT. For more information, see “Specifying Quoted Strings” on page 3-33 .
Constraint name	Constraint name	An example of an SQL statement that creates a constraint name is CREATE TABLE.
Cursor name	Identifier	An example of an SQL statement that creates a cursor name is DECLARE. For more information, see “Using Non-ASCII Characters in Source Code” on page 7-6 .

(1 of 3)

SQL Identifier	SQL Segment	Notes
Database name	Database name	<p>An example of an SQL statement that creates a database name is CREATE DATABASE</p> <p>Restrictions apply when you use multibyte characters for database names in SE. For more information, see “Generating Non-ASCII Filenames” on page 6-4.</p>
Filename	None	<p>An example of an SQL statement that specifies a pathname and filename is LOAD.</p> <p>The syntax for pathnames and filenames (including log files) depends on the operating system.</p> <p>If you use multibyte characters in pathnames, you limit portability of the files to those operating systems that can support multibyte filenames. For more information, see “Generating Non-ASCII Filenames” on page 5-4, page 6-4, and page 7-4.</p>
Host variable name	None	<p>An example of an SQL statement that specifies a host variable is FETCH.</p> <p>For more information, see “Using Non-ASCII Characters in Source Code” on page 7-6.</p>
Index name	Index name	<p>An example of an SQL statement that creates an index name is CREATE INDEX.</p>
Role name	Identifier	<p>An example of an SQL statement that creates a role name is CREATE ROLE.</p> <p>You cannot execute statements that are related to roles in SE.</p>
Statement identifier	Identifier	<p>An example of an SQL statement that creates a statement identifier is PREPARE.</p> <p>For more information, see “Using Non-ASCII Characters in Source Code” on page 7-6.</p>
Stored procedure name	Procedure name	<p>An example of an SQL statement that creates a stored procedure name is CREATE PROCEDURE.</p>

(2 of 3)

SQL Identifier	SQL Segment	Notes
Stored procedure variable name	Expression	An example of an SQL statement that creates a stored procedure variable name is CREATE PROCEDURE.
Synonym name	Synonym name	An example of an SQL statement that creates a synonym name is CREATE SYNONYM.
Table name	Table name	An example of an SQL statement that creates a table name is CREATE TABLE. Restrictions apply when you use multibyte characters for table names in SE. For more information, see “Generating Non-ASCII Filenames” on page 6-4 .
Trigger correlation name	Identifier	An example of an SQL statement that creates a trigger correlation name is CREATE TRIGGER.
Trigger name	Identifier	An example of an SQL statement that creates a trigger name is CREATE TRIGGER.
View name	View name	An example of an SQL statement that creates a view name is CREATE VIEW.

(3 of 3)

References to SQL Segments

The SQL Segment column in [Figure 3-1 on page 3-5](#) refers to the segment in the [Informix Guide to SQL: Syntax](#) that gives the complete syntax of the identifier. In many cases, the complete syntax of an SQL segment can include other identifiers. For example, the Index Name segment in the [Informix Guide to SQL: Syntax](#) shows that the syntax of an index name can include a database name, a database server name, and an owner name as well as the simple name of the index.

When you look up a particular object in [Figure 3-1](#), keep in mind that the simple name of the object accepts multibyte characters, but the other identifiers in the syntax for that object accept multibyte characters only if they also appear in the table. For example, the database name identifier within the Index Name segment accepts multibyte characters, but the database-server-name identifier within the Index Name segment does not accept multibyte characters.

ANSI

Owner Name in SQL Identifiers

The owner name provides further identification of a database object within a database.

The ANSI term for an owner name is a schema name. ♦

The ability to put non-ASCII characters in the owner-name portion of an identifier depends on whether your operating system supports multibyte characters in user names.

UNIX

If your database server is on a computer with the UNIX operating system, the owner-name qualifier defaults to the UNIX login ID. However, most versions of UNIX do not support multibyte characters in the UNIX login IDs. ♦

You can use multibyte characters in owner names if you explicitly specify an owner name (in single quotes) when you create database objects. For example, you can assign an owner name that contains multibyte characters when you put the owner-name portion of the index name in quotes in a CREATE INDEX statement.



Warning: *If you specify multibyte characters in an owner name on a UNIX system, you do so at your own risk. If a UNIX login ID is used to match the owner name, the match might fail.*

The following example shows a CREATE INDEX statement that specifies a multibyte owner name. In this example, the owner name consists of three 2-byte characters:

```
CREATE INDEX 'A1A2B1B2C1C2'.myidx ON mytable (mycol)
```

The preceding example assumes that the client locale supports a multibyte code set and that A¹A², B¹B², and C¹C² are valid characters in this code set.

Valid Characters in Identifier Names

In the syntax of an SQL identifier, a *letter* can be any character that the alpha class of the locale defines. The alpha class lists all characters that are classified as alphabetic. (For further information on character classification, see [“The CTYPE Category” on page A-6.](#)) In the default locale, the alpha class of the code set includes the ASCII characters in the ranges a to z and A to Z. When Informix products use the default locale, SQL identifiers can use these ASCII characters wherever *letter* appears in the syntax of an SQL identifier.

In a nondefault locale, the alpha class of the locale also lists the ASCII characters in the ranges a to z and A to Z. It might also include non-ASCII characters such as non-ASCII digits or ideographic characters. For example, the alpha class of the Japanese UJIS code set (in the Japanese UJIS locale) contains Kanji characters. When Informix products use a nondefault locale, SQL identifiers can use ASCII characters wherever *letter* is valid in the syntax of an SQL identifier. A non-ASCII character is also valid for *letter* as long as this character is listed in the alpha class of the locale.

The SQL statements in the following example use non-ASCII characters as letters in SQL identifiers:

```
CREATE DATABASE marché;

CREATE TABLE équipement
(
  code NCHAR(6),
  description NVARCHAR(128,10),
  prix_courant MONEY(6,2)
);

CREATE VIEW çà_va AS
  SELECT numéro,nom FROM abonnés;
```

In this example, the user creates the following database, table, and view with French-language character names in a French locale (such as **fr_fr.8859-1**):

- The CREATE DATABASE statement uses the identifier **marché**, which includes the 8-bit character **é**, to name the database.
- The CREATE TABLE statement uses the identifier **équipement**, which includes the 8-bit character **é**, to name the table, and the identifiers **code**, **description**, and **prix_courant** to name the columns.
- The CREATE VIEW statement uses the identifier **ça_va**, which includes the 8-bit characters **ç** and **à**, to name the view.
- The SELECT clause within the CREATE VIEW statement uses the identifiers **numéro** and **nom** for the columns in the select list and the identifier **abonnés** for the table in the FROM clause. Both **numéro** and **abonnés** include the 8-bit character **é**.

All of the identifiers in this example conform to the rules for specifying identifiers. For these names to be valid, the client locale must support a code set with these French characters.

For the complete syntax and usage of identifiers in SQL statements, see the Identifier segment in the [Informix Guide to SQL: Syntax](#).

Non-ASCII Characters in Delimited Identifiers

A delimited identifier is an identifier that is enclosed in double quotes. When the **DELIMIDENT** environment variable is set, the database server interprets sequences of characters in double quotes as delimited identifiers and sequences of characters in single quotes as strings. This interpretation of quotes is compliant with the ANSI standard.

When you use a nondefault locale, you can specify non-ASCII characters in most delimited identifiers. You can put non-ASCII characters in a delimited identifier if you can put non-ASCII characters in the undelimited form of the same identifier. For example, [Figure 3-1 on page 3-5](#) shows that you can put non-ASCII characters in an undelimited index name. Thus you can put non-ASCII characters in an index name that you have enclosed in double quotes to make it a delimited identifier, as follows:

```
CREATE INDEX "A1A2#B1B2" ON mytable (mycol)
```

For the complete description of delimited identifiers, see the Identifier segment in the [Informix Guide to SQL: Syntax](#).

Multibyte Characters and Identifier Length

An SQL database server limits the name of an SQL identifier to 18 bytes. When you use multibyte characters in identifier names, you must ensure that the identifier does not exceed this size requirement.



Tip: The name of an SE database is limited to 10 bytes. For more information, see [“Naming a Database” on page 6-7](#).

For example, the following CREATE SYNONYM statement creates a synonym name of 8 multibyte characters:

```
CREATE SYNONYM A1A2A3B1B2C1C2C3D1D2E1E2F1F2G1G2H1H2 FOR A1A2B1B2
```

The synonym name shown in the preceding example is 18 bytes long (six 2-byte multibyte characters and two 3-byte multibyte characters), so it does not exceed the maximum length for identifier names. However, the following `CREATE SYNONYM` statement generates an error because the total number of bytes in this synonym name is 20:

```
CREATE SYNONYM A1A2A3B1B2B3C1C2C3D1D2D3E1E2F1F2G1G2H1H2 FOR A1A2B1B2
```

This statement specifies four 3-byte characters and four 2-byte characters for the synonym name. Even though the synonym name has only eight characters, the total number of bytes in the synonym name would be 20 bytes, which exceeds the maximum length for an identifier name.

Using Character Data Types

The choice of a locale can affect the way that the database server handles character data. The section explains how this choice affects data in the following SQL character data types:

- Locale-sensitive character data types: `NCHAR` and `NVARCHAR`
- Other character data types: `CHAR`, `VARCHAR`, and `TEXT`

You specify data types in data definition statements such as `CREATE TABLE` and `ALTER TABLE`.

The NCHAR Data Type

The `NCHAR` data type stores character data in a fixed-length field. This data can be a sequence of single-byte or multibyte letters, numbers, and symbols. However, the code set of your database locale must support this character data. `NCHAR` columns typically store names, addresses, phone numbers, and so on.

The syntax of the `NCHAR` data type is as follows:

```
NCHAR(size)
```

In this format, the *size* parameter specifies the number of bytes in the column. The total length of an NCHAR column cannot exceed 32,767 bytes for Universal Server and OnLine Dynamic Server or 32,511 bytes for SE. If you do not specify *size*, the database server assumes NCHAR(1). For the complete syntax of the NCHAR data type, see the Data Type section in the [Informix Guide to SQL: Syntax](#).

Because the length of this column is fixed, when the database server retrieves or sends an NCHAR value, it transfers exactly *size* bytes of data. If the length of a character string is shorter than *size*, the database server extends the string with spaces to make up the *size* bytes. If the string is longer than *size* bytes, the database server truncates the string.

Collating NCHAR Data

The NCHAR data type is a locale-sensitive data type. The only difference between NCHAR and CHAR data types is the collation order. The database server collates data in NCHAR columns in localized order. The database server collates data in CHAR columns in code-set order. For more information on collation order, see [“The Collation Order” on page 1-10](#).

Tip: *The default locale (U.S. English) does not specify a localized order. Therefore, the database server sorts NCHAR data in code-set order. When you use the default locale, there is no difference between CHAR and NCHAR data.*



Handling NCHAR Data

Within a client application, always manipulate NCHAR data in the **CLIENT_LOCALE** of the client application. The client application performs code-set conversion of NCHAR data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**. For information on code-set conversion, see [page 1-40](#).

Multibyte Characters with NCHAR

To store multibyte character data in an NCHAR column, your database locale must support a code set with these same multibyte characters. When you store multibyte characters, make sure to calculate the number of bytes that are needed. The *size* parameter of the NCHAR data type refers to the number of bytes of storage that is reserved for the data.

Because one multibyte character might use several bytes for storage, the value of *size* bytes does not indicate the number of characters that the column can hold. The total number of multibyte characters that you can store in the column is less than the total number of bytes that you can store in the column. Make sure to declare the *size* value of the NCHAR column in such a way that it can hold enough characters for your purposes.

Treating NCHAR Values as Numeric Values

If you plan to perform calculations on numbers that are stored in a column, assign a numeric data type (such as INTEGER or FLOAT) to that column. The description of the CHAR data type in the [Informix Guide to SQL: Reference](#) gives detailed reasons why you should not treat certain numbers as CHAR values. The same reasons apply for certain numeric values as NCHAR values. Treat only numbers that have leading zeros (such as postal codes) as NCHAR data types. Use NCHAR only if you need to sort the numeric values in localized order.

Nonprintable Characters with NCHAR

An NCHAR value can include tabs, spaces, and nonprintable characters. Nonprintable NCHAR and CHAR values are entered, displayed, and treated similarly. For more information on nonprintable characters, see the description of the CHAR data type in the [Informix Guide to SQL: Reference](#).

The NVARCHAR Data Type

The NVARCHAR data type stores character data in a variable-length field. This data can be a sequence of single-byte or multibyte letters, numbers, and symbols. However, the code set of your database locale must support this character data.

The syntax of the NVARCHAR data type is as follows:

```
NVARCHAR(max, reserve)
```

In this format, the *max* parameter specifies the maximum number of bytes that can be stored in the column, and the *reserve* parameter specifies the minimum number of bytes that are reserved for the column. You must specify the maximum size (*max*) of the NVARCHAR column. The size of this parameter cannot exceed 255 bytes. When you place an index on an NVARCHAR column, the maximum size is 254 bytes. You can store shorter, but not longer, character strings than the value that you specify.

You can optionally specify the minimum reserved space (*reserve*) parameter. This value can range from 0 to 255 bytes but must be less than the maximum size (*max*) of the NVARCHAR column. If you do not specify a minimum space value, the default value of *reserve* is 0. Specify the *reserve* parameter when you initially intend to insert rows with short or null data in this column but later expect the data to be updated with longer values.

Although use of NVARCHAR economizes on space that is used in a table, it has no effect on the size of an index. In an index that is based on an NVARCHAR column, each index key has a length equal to *max* bytes, the maximum size of the column.

The database server does not strip an NVARCHAR object of any user-entered trailing spaces, nor does it pad the NVARCHAR object to the full length of the column. However, if you specify a minimum reserved space (*reserve*), and some of the data values are shorter than that amount, some of the space that is reserved for rows goes unused.

For the complete syntax of the NVARCHAR data type, see the Data Type segment in the [Informix Guide to SQL: Syntax](#).



Important: The INFORMIX-SE database server does not support the NVARCHAR data type.

Collating NVARCHAR Data

The NVARCHAR data type is a locale-sensitive data type. The only difference between NVARCHAR and VARCHAR data types is the collation order. The database server collates data in NVARCHAR columns in localized order. The database server collates data in VARCHAR columns in code-set order. For more information on collation order, see [“The Collation Order” on page 1-10](#).



Tip: The default locale (U.S. English) does not specify a localized order. Therefore, the database server sorts NVARCHAR data in code-set order. When you use the default locale, there is no difference between VARCHAR and NVARCHAR data.

Handling NVARCHAR Data

Within a client application, always manipulate NVARCHAR data in the **CLIENT_LOCALE** of the client application. The client application performs code-set conversion of NVARCHAR data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**. For information on code-set conversion, see [page 1-40](#).

Multibyte Characters with NVARCHAR

To store multibyte character data in an NVARCHAR column, your database locale must support a code set with these same multibyte characters. When you store multibyte characters, make sure to calculate the number of bytes that are needed. The *max* parameter of the NVARCHAR data type refers to the maximum number of bytes that the column can store.

Because one multibyte character might use several bytes for storage, the value of *max* bytes does not indicate the number of characters that the column can hold. The total number of multibyte characters that you can store in the column is less than the total number of bytes that you can store in the column. Make sure to declare the *max* value of the NVARCHAR column so that it can hold enough characters for your purposes.

Nonprintable Characters with NVARCHAR

An NVARCHAR value can include tabs, spaces, and nonprintable characters. Nonprintable NVARCHAR characters are entered, displayed, and treated in the same way as nonprintable VARCHAR characters. For detailed information on how to enter and display nonprintable characters, see the description of the VARCHAR data type in the [Informix Guide to SQL: Reference](#).



Tip: The database server interprets the null character (ASCII 0) as a C null terminator. Therefore, in NVARCHAR data, the null terminator acts as a string-terminator character.

Storing Numeric Values in an NVARCHAR Column

When you insert a numeric value into a NVARCHAR column, the database server does not pad the value with trailing blanks up to the maximum length of the column. The number of digits in a numeric NVARCHAR value is the number of characters that you need to store that value. For example, the database server stores a value of 1 in the **mytab** table when it executes the following SQL statements:

```
CREATE TABLE mytab (col1 VARCHAR(10));  
INSERT INTO mytab VALUES (1);
```

Performance Considerations for NCHAR and NVARCHAR

The NCHAR data type is very similar to the CHAR data type, and NVARCHAR is very similar to the VARCHAR data type. The difference between these data types is as follows:

- The database server collates NCHAR and NVARCHAR column values in localized order.
- The database server collates CHAR and VARCHAR column values in code-set order.

Localized collation is dependent on the sorting rules that the locale defines, not simply on the computer representation of the character (the code points). This difference means that the database server might perform complex processing to compare and collate NCHAR and NVARCHAR data. Therefore, access to NCHAR data might be slower with respect to comparison and collation than to CHAR data. Similarly, access to data in an NVARCHAR column might be slower with respect to comparison and collation than access to the same data in a VARCHAR column.

Assess whether your character data needs to take advantage of localized order for collation and comparison. If code-set order is adequate, use the CHAR and the VARCHAR data types.

Other Character Data Types

The choice of locale can affect the character data types CHAR, VARCHAR and TEXT. This section describes how this choice affects each of these character data types.

The CHAR Data Type

The CHAR data type stores character data in a fixed-length field. This data can consist of letters, numbers, and symbols. The following list summarizes how choice of a locale affects the CHAR data type:

- The size of a CHAR column is byte based, not character based.
For example, if you define a CHAR column as CHAR(10), the column has a fixed length of 10 bytes, not 10 characters. If you want to store multibyte characters in a CHAR column, keep in mind that the total number of characters you can store in the column might be less than the total number of bytes you can store in the column. Make sure to define the byte size of the CHAR column so that it can hold enough characters for your purposes.
- You can enter single-byte or multibyte characters in a CHAR column.
The database locale must support the characters that you want to store in CHAR columns.
- The database server sorts CHAR columns in code-set order, not in localized order.
For more information on collation order, see [“The Collation Order” on page 1-10](#).
- Within a client application, always manipulate CHAR data in the **CLIENT_LOCALE** of the client application.
The client application performs code-set conversion of CHAR data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**. For information on code-set conversion, see [page 1-40](#).

For the syntax of the CHAR data type, see the [Informix Guide to SQL: Syntax](#). For a detailed explanation of the CHAR data type, see the [Informix Guide to SQL: Reference](#).

The VARCHAR Data Type

The VARCHAR data type stores character data in a variable-length field. This data can consist of letters, numbers, and symbols. The following list summarizes how the choice of a locale affects the VARCHAR data type:

- The maximum size and minimum reserved space for a VARCHAR column are byte based, not character based.
For example, if you define a VARCHAR column as VARCHAR(10,6), the column has a maximum length of 10 bytes and a minimum reserved space of 6 bytes. If you want to store multibyte characters in a VARCHAR column, keep in mind that the total number of characters you can store in the column might be less than the total number of bytes you can store in the column. Make sure to define the maximum byte size of the VARCHAR column so that it can hold enough characters for your purposes.
- You can enter single-byte or multibyte characters in a VARCHAR column.

The database locale must support the characters that you want to store in VARCHAR columns.

- The database server sorts VARCHAR columns in code-set order, not in localized order.

For more information on collation order, see [“The Collation Order” on page 1-10](#).

- Within a client application, always manipulate VARCHAR data in the **CLIENT_LOCALE** of the client application.

The client application performs code-set conversion of VARCHAR data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**. For information on code-set conversion, see [page 1-40](#).

For the syntax of the VARCHAR data type, see the [Informix Guide to SQL: Syntax](#). For a detailed explanation of the VARCHAR data type, see the [Informix Guide to SQL: Reference](#).

The TEXT Data Type

The TEXT data type stores any kind of text data. TEXT columns typically store memos, manual chapters, business documents, program source files, and other types of textual information. The following list summarizes how the choice of a locale affects the TEXT data type:

- The database server stores character data in a TEXT column in the code set of the database locale.
- You can enter single-byte or multibyte characters in a TEXT column. The database locale should support the characters that you want to store in TEXT columns. However, you can put any type of character in a TEXT column.
- Text columns do not have an associated collation order. The database server does not build indexes on TEXT columns. Therefore, it does not perform collation tasks on these columns. For more information on collation order, see [“The Collation Order” on page 1-10](#).
- Within a client application, always manipulate TEXT data in the **CLIENT_LOCALE** of the client application. The client application performs code-set conversion of TEXT data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**. For information on code-set conversion, see [page 1-40](#).

For the complete syntax of the TEXT data type, see the [Informix Guide to SQL: Syntax](#). For a detailed explanation of the TEXT data type, see Chapter 3 of the [Informix Guide to SQL: Reference](#).

Collating Character Data

Collation involves the sorting of the data values in columns that have character data types. For an explanation of collation order and a discussion of the two methods of sorting character data (code-set order and localized order), see [“The Collation Order” on page 1-10](#).

The type of collation order that the database server uses affects the following SQL statements:

- CREATE INDEX
- SELECT

Collation Order in CREATE INDEX

The CREATE INDEX statement creates an index on one or more columns of a table. The ASC and DESC keywords in the CREATE INDEX statement control whether the index keys are stored in ascending or descending order.

When you use a nondefault locale, the following locale-specific considerations apply to the CREATE INDEX statement:

- When you create an index on a CHAR or VARCHAR column, the index keys are stored in code-set order.
For example, if the database stores its database locale as the Japanese SJIS locale (**ja_jp.sjis**), index keys for a CHAR column in any table of the database are stored in Japanese SJIS code-set order.
- When you create an index on an NCHAR or NVARCHAR column, the index keys are stored in localized order.
For example, if the database is branded with the Japanese SJIS locale, index keys for an NCHAR column in any table of the database are stored in the localized order that the **ja_jp.sjis** locale defines.

If you use the default locale (U.S. English), the index keys are stored in the code-set order (in ascending or descending sequence) of the default code set regardless of the data type of the character column. Because the default locale does not define a localized order, the database server sorts NCHAR and NVARCHAR columns, as well as CHAR and VARCHAR columns, in code-set order.

For a complete description of the CREATE INDEX statement, see the [Informix Guide to SQL: Syntax](#).

Collation Order in *SELECT* Statements

The *SELECT* statement performs a query on the specified table and retrieves data from the specified columns and rows. Collation order affects the following parts of the *SELECT* statement:

- The *ORDER BY* clause
- The relational operator, *BETWEEN*, and *IN* conditions of the *WHERE* clause
- The *MATCHES* and *LIKE* conditions of the *WHERE* clause

For a complete description of the *SELECT* statement, see the [Informix Guide to SQL: Syntax](#).

The ORDER BY Clause

The *ORDER BY* clause sorts the retrieved rows by the values that are contained in a column or set of columns. When this clause sorts character columns, the results of the sort depend on the data type of the column, as follows:

- *CHAR* and *VARCHAR* columns are sorted in code-set order.
- *NCHAR* and *NVARCHAR* columns are sorted in localized order.

Assume that you use a nondefault locale for the client and database locale, and you make a query against the table called **abonnés**. The following *SELECT* statement specifies three columns of *CHAR* data type in the select list: **numéro** (employee number), **nom** (last name), and **prénom** (first name):

```
SELECT numéro,nom,prénom
FROM abonnés
ORDER BY nom;
```

The statement sorts the query results by the values that are contained in the **nom** column. Because the **nom** column that is specified in the ORDER BY clause is a CHAR column, the database server sorts the query results in the code-set order. As Figure 3-2 shows, names that begin with uppercase letters come before names that begin with lowercase letters, and names that start with an accented letter (Ålesund, Étaix, Ötker, and Øverst) come at the end of the list.

Figure 3-2
Data Set for Code-Set Order of the abonné Table

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13606	Dupré	Michèle Françoise
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13609	Tiramisù	Paolo Alfredo
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13601	Ålesund	Sverre
13608	Étaix	Émile
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders

However, the result of the query is different if the **numéro**, **nom**, and **prénom** columns of the **abonnés** table are defined as NCHAR rather than CHAR.

Suppose the nondefault locale defines a localized order that collates the data as Figure 3-3 shows. This localized order defines equivalence classes for uppercase and lowercase letters and for unaccented and accented versions of the same letter.

Figure 3-3
Data Set for Localized Order of the abonné Table

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13601	Ålesund	Sverre
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13606	Dupré	Michèle Françoise
13608	Étaix	Émile
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaitre	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders
13609	Tiramisù	Paolo Alfredo

The same *SELECT* statement now returns the query results in localized order because the **nom** column that the *ORDER BY* clause specifies is an *NCHAR* column.

The SELECT statement supports use of a column substring in an ORDER BY clause. However, you need to ensure that this use for column substrings works with the code set that your locale supports. For more information, see [“Partial Characters in Column Substrings” on page 3-36](#).

Logical Predicates in a WHERE Clause

The WHERE clause specifies search criteria and join conditions on the data that you want to select. Collation rules affect the WHERE clause when the expressions in the condition are column expressions with character data types and the search condition is one of the following logical predicates:

- Relational-operator condition
- BETWEEN condition
- IN condition
- EXISTS and ANY conditions

Relational-Operator Conditions

The following SELECT statement assumes a nondefault locale. It uses the less than (<) relational operator to specify that only those rows are to be retrieved from the **abonnés** table in which the value of the **nom** column is less than Hammer.

```
SELECT numéro,nom,prénom
FROM abonnés
WHERE nom < 'Hammer';
```

If **nom** is a CHAR column, the database server uses code-set order of the default code set to retrieve the rows that the WHERE clause specifies. The following sample of output shows that this SELECT statement retrieves only two rows.

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13606	Dupré	Michèle Françoise

These two rows are those less than `Hammer` in the code-set-ordered data set shown in [Figure 3-2 on page 3-22](#).

However, if **nom** is an NCHAR column, the database server uses localized order to sort the rows that the WHERE clause specifies. The following sample of output shows that this SELECT statement retrieves six rows.

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13601	Ålesund	Sverre
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13606	Dupré	Michèle Françoise
13608	Étaix	Émile

These six rows are those less than `Hammer` in the localized-order data set shown in [Figure 3-3 on page 3-23](#).

BETWEEN Conditions

The following SELECT statement assumes a nondefault locale and uses a BETWEEN condition to retrieve only those rows in which the values of the **nom** column are in the inclusive range of the values of the two expressions that follow the BETWEEN keyword:

```
SELECT numéro,nom,prénom
FROM abonnés
WHERE nom BETWEEN 'A' AND 'Z';
```

The query result depends on whether **nom** is a CHAR or NCHAR column. If **nom** is a CHAR column, the database server uses the code-set order of the default code set to retrieve the rows that the WHERE clause specifies. The following sample output shows the query results.

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13606	Dupré	Michèle Françoise
13607	Hammer	Gerhard

(1 of 2)

numéro	nom	prénom
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaitre	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13609	Tiramisù	Paolo Alfredo

(2 of 2)

Because the database server uses the code-set order for the **nom** values (see [Figure 3-2 on page 3-22](#)), these query results do not include the following rows:

- Rows in which the value of **nom** begins with a lowercase letter:
da Sousa and di Girolamo
- Rows with an accented letter: Ålesund, Étaix, Ötker, and Øverst

However, if **nom** is an NCHAR column, the database server uses localized order to sort the rows. The following sample output shows the query results.

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13601	Ålesund	Sverre
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13606	Dupré	Michèle Françoise
13608	Étaix	Émile
13607	Hammer	Gerhard
13602	Hämmerle	Greta

(1 of 2)

numéro	nom	prénom
13604	LaForêt	Jean-Noël
13610	LeMaitre	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders
13609	Tiramisù	Paolo Alfredo

(2 of 2)

Because the database server uses localized order for the **nom** values, these query results include rows in which the value of **nom** begins with a lowercase letter or accented letter.

IN Conditions

An IN condition is satisfied when the expression to the left of the IN keyword is included in the parenthetical list of values to the right of the keyword. The following SELECT statement assumes a nondefault locale and uses an IN condition to retrieve only those rows in which the value of the **nom** column is any of the following: Azevedo, Llanero, or Oatfield.

```
SELECT numéro,nom,prénom
FROM abonnés
WHERE nom IN ('Azevedo', 'Llanero', 'Oatfield');
```

The query result depends on whether **nom** is a CHAR or NCHAR column. If **nom** is a CHAR column, the database server uses code-set order (see [Figure 3-2 on page 3-22](#)). The database server retrieves rows in which the value of **nom** is Azevedo, but not rows in which the value of **nom** is azevedo or Åzevedo because the characters A, a, and Å are not equivalent in the code-set order (see [Figure 3-2](#)). The query also returns rows with the **nom** values of Llanero and Oatfield.

However, if **nom** is an NCHAR column, the database server uses localized order (see [Figure 3-3 on page 3-23](#)) to sort the rows. If the locale defines A, a, and Å as equivalent characters in the localized order, the query returns rows in which the value of **nom** is Azevedo, azevedo, or Åzevedo. The same selection rule applies to the other names in the parenthetical list that follows the IN keyword.

Comparisons with MATCHES and LIKE Conditions

Collation rules also affect the WHERE clause when the expressions in the condition are column expressions with character data types and the search condition is one of the following conditions:

- MATCHES condition
- LIKE condition

MATCHES Condition

A MATCHES condition tests for matching character strings. The condition is true, or satisfied, when the value of the column to the left of the MATCHES keyword matches the pattern that a quoted string specifies to the right of the MATCHES keyword. You can use wildcard characters in the string. For example, you can use brackets to specify a range of characters. (For more information on MATCHES, see the Condition Segment in Chapter 1 of the [Informix Guide to SQL: Syntax](#).)

When the MATCHES condition does not list a range of characters in the string, it specifies a *literal match*. For literal matches, the data type of the column determines whether collation considerations come into play, as follows:

- For CHAR and VARCHAR columns, no collation considerations come into play.
- For NCHAR and NVARCHAR columns, collation considerations might come into play because these data types use localized order and the locale might define equivalence classes of collation.

For example, the localized order might specify that a and A are an equivalent class. That is, they have equal weight in the collation sequence. For further information on localized order, see [page 1-11](#).

The examples in the following table illustrate the different results that CHAR and NCHAR columns produce when a user specifies the MATCHES keyword without a range in a SELECT statement. These examples assume use of a nondefault locale that defines A and a in an equivalence class. It also assumes that **col1** is a CHAR column and **col2** is an NCHAR column in table **mytable**.

Query	Data Type	Query Results
SELECT * FROM mytable WHERE col1 MATCHES 'art'	CHAR	All rows in which column col1 contains the value 'art' with a lowercase a
SELECT * FROM mytable WHERE col2 MATCHES 'art'	NCHAR	All rows in which column col2 contains the value 'art' or 'Art'

When you use the MATCHES keyword to specify a range, collation considerations come into play for all columns with character data types. When the column to the left of the MATCHES keyword is an NCHAR, NVARCHAR, CHAR, or VARCHAR column, and the quoted string to the right of the MATCHES keyword includes brackets to specify a range, the database server uses localized order.



Important: When the database server determines the characters that fall within a range, it always uses the localized order that is specified for the database, even for CHAR and VARCHAR columns. This behavior is an exception to the rule that the database server uses code-set order for all operations on CHAR and VARCHAR columns and uses localized order for all operations on NCHAR and NVARCHAR columns.

Some simple examples show how the database server treats NCHAR, NVARCHAR, CHAR, and VARCHAR columns when you use the MATCHES keyword with a range in a SELECT statement. Suppose that you want to retrieve from the **abonnés** table the employee number, first name, and last name for all employees whose last name **nom** begins in the range of characters E through P. Also assume that the **nom** column is an NCHAR column. The following SELECT statement uses a MATCHES condition in the WHERE clause to pose this query:

```
SELECT numéro,nom,prénom
FROM abonnés
WHERE nom MATCHES '[E-P]*'
ORDER BY nom;
```

The rows for Étaix, Ötker, and Øverst appear in the query result because, in the localized order ([Figure 3-3 on page 3-23](#)), the accented first letter of each name falls within the E through P MATCHES range for the **nom** column.

numéro	nom	prénom
13608	Étaix	Émile
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders

If **nom** is a CHAR column, the query result is exactly the same as when **nom** was an NCHAR column. The database server always uses localized order to determine what characters fall within a range, regardless of whether the column is CHAR or NCHAR.

LIKE Condition

A LIKE condition also tests for matching character strings. As with the MATCHES condition, the LIKE condition is true, or satisfied, when the value of the column to the left of the LIKE keyword matches the pattern that the quoted string specifies to the right of the LIKE keyword. You can only use certain symbols as wildcards in the quoted string. (For more information on LIKE, see the Condition Segment in Chapter 1 of the [Informix Guide to SQL: Syntax](#).)

The LIKE condition can only specify a *literal match*. For literal matches, the data type of the column determines whether collation considerations come into play, as follows:

- For CHAR and VARCHAR columns, no collation considerations come into play.
- For NCHAR and NVARCHAR columns, collation considerations might come into play because these data types use localized order, and the locale might define equivalence classes of collation.

For example, the localized order might specify that a and A are an equivalent class. For more information, see the description of literal matches with the MATCHES condition on [page 3-28](#).

The LIKE keyword does not support matches with a range. That is, you cannot use bracketed wildcard characters in LIKE conditions.

Wildcard Characters in LIKE and MATCHES Conditions

Informix products support the following ASCII characters as wildcard characters in the MATCHES and LIKE conditions.

Clause	Wildcard Characters
LIKE	_ %
MATCHES	* ? [] ^ -

For CHAR and VARCHAR data, the database server performs byte-by-byte comparison for pattern matching in the LIKE and MATCHES conditions. For NCHAR and NVARCHAR data, the database server performs pattern matching in the LIKE and MATCHES conditions based on characters, not bytes. Therefore, the `_` (underscore) wildcard of the LIKE clause and the `?` wildcard of the MATCHES clause match any one single-byte or multibyte character, as the following table shows.

Condition	Quoted String	Column Value	Result
LIKE	'ab_d'	'abcd'	True
LIKE	'ab_d'	'abA ¹ A ² d'	True
MATCHES	'ab?d'	'abcd'	True
MATCHES	'ab?d'	'abA ¹ A ² d'	True

The database server treats any multibyte character as a literal character. To tell the database server to interpret a wildcard character as its literal meaning, you must precede the character with an escape character. You must use single-byte characters as escape characters; the database server does not recognize use of multibyte characters for this purpose. The default escape character is the backslash (`\`).

The following use of the MATCHES clause gives a true result for the column value that is shown.

Clause	Quoted String	Column Value	Result
MATCHES	'ab\?d'	'ab?d'	True

Handling Character Data

The GLS feature allows you to put non-ASCII characters (including multibyte characters) in the following parts of an SQL statement:

- Quoted strings
- Comments
- Column substrings
- TRIM function arguments

Specifying Quoted Strings

You use quoted strings in a variety of SQL statements, particularly data manipulation statements such as SELECT and INSERT. A quoted string is a sequence of characters that are delimited by quotation marks. The quotation marks can be single quotes or double quotes. However, if the **DELIMIDENT** environment variable is set, the database server interprets a sequence of characters in double quotes as a delimited identifier rather than as a string. (For more information about delimited identifiers, see [page 3-10](#).)

When you use a nondefault locale, you can use any characters in the code set of your locale within a quoted string. If the locale supports a code set with non-ASCII characters, you can use these characters in a quoted string. In the following example, the user inserts column values that include multibyte characters into the table **mytable**:

```
INSERT INTO mytable
VALUES ('A1A2B1B2abcd', '123X1X2Y1Y2', 'efgh')
```

In this example, the first quoted string includes the multibyte characters A¹A² and B¹B². The second quoted string includes the multibyte characters X¹X² and Y¹Y². The third quoted string contains only single-byte characters. This example assumes that the locale supports a multibyte code set with the A¹A², B¹B², X¹X², and Y¹Y² characters.

For complete information on quoted strings, see the Quoted String segment in the [Informix Guide to SQL: Syntax](#).

Specifying Comments

To use comments after SQL statements, introduce the comment text with one of the following comment symbols:

- The double dash (--) complies with the ANSI SQL standard.
- Curly brackets ({}), are an Informix extension to the standard.

When you use a nondefault locale, you can use any characters in the code set of your locale within a comment. If the locale supports a code set with non-ASCII characters, you can use these characters in an SQL comment. In the following example, the user inserts a column value that includes multibyte characters into the table **mytable**:

```
EXEC SQL insert into mytable  
values ('A1A2B1B2abcd', '123') -- A1A2 and B1B2 are multibyte characters.
```

In this example, the SQL comment includes the multibyte characters A¹A² and B¹B². This example assumes that the locale supports a multibyte code set with the A¹A² and B¹B² characters.

For complete information on SQL comments and comment symbols, see the Introduction to the [Informix Guide to SQL: Syntax](#).

Specifying Column Substrings

When you specify a column expression with a character data type in a SELECT statement (or in any other SQL statement that includes an embedded SELECT statement), you can specify that a subset of the data in the column is to be retrieved. A column expression that includes brackets to signify a subset of the data in the column is known as a *column substring*. The syntax of a column substring is as follows:

```
column_name[first, last]
```

In this format, *first* is the position of the first byte in the substring, and *last* is the position of the last byte in the substring. For the complete syntax of column expressions, see the Expression segment in the [Informix Guide to SQL: Syntax](#).

Column Substrings in Single-Byte Code Sets

Suppose that you want to retrieve the **customer_num** column and the seventh through ninth bytes of the **lname** column from the **customer** table. To perform this query, use a column substring for the **lname** column in your **SELECT** statement, as follows:

```
SELECT customer_num, lname[7,9] as lname_subset
FROM customer
WHERE lname = 'Albertson'
```

If the value of the **lname** column is **Albertson**, the following sample output shows the result of the query.

customer_num	lname_subset
114	son

Because the locale supports a single-byte code set, the preceding query seems to return the seventh through ninth characters of the name **Albertson**. However, column substrings are byte based, and the query returns the seventh through ninth bytes of the name **Albertson**. Because one byte is equal to one character in single-byte code sets, the distinction between characters and bytes in column substrings is not apparent in these code sets.

Column Substrings in Multibyte Code Sets

For multibyte code sets, column substrings return the specified number of bytes, not number of characters. If a character column **multi_col** contains a string that consists of three 2-byte characters, this 6-byte string can be represented as follows:

A¹A²B¹B²C¹C²

Suppose that you specified the following column substring for the **multi_col** column in a query:

multi_col[1,2]

The query returns the following result:

A¹A²

The substring that the query returns consists of 2 bytes (1 character), not two characters.

To retrieve the first two characters from the **multi_col** column, specify a column substring in which *first* is the byte position of the first byte in the first character and *last* is the byte position of the last byte in the second character. For the 6-byte string A¹A²B¹B²C¹C², you specify this column substring as follows in your query:

```
multi_col[1,4]
```

The following result is returned:

```
A1A2B1B2
```

The substring that the query returns consists of the first 4 bytes of the column value as you specified. These 4 bytes represent the first two characters in the column.

Partial Characters in Column Substrings

A multibyte character might consist of 2, 3, or 4 bytes. A multibyte character that has lost one or more of its bytes so that the original intended meaning of the character is lost is called a *partial character*.

Unless prevented, a column substring might truncate a multibyte character or split it up in such a manner that it no longer retains the original sequence of bytes. A partial character might be generated when you use column subscript operators on columns that contain multibyte characters. Suppose that a user specifies the following column substring for the **multi_col** column where the value of the string in **multi_col** is A¹A²B¹B²C¹C²:

```
multi_col[2,5]
```

The user requests the following bytes in the query: A²B¹B²C¹. However, if the database server returned this column substring to the user, the first and third characters in the column would be truncated.

Avoidance in a Multibyte Code Set

Informix database servers do not allow partial characters to occur. The GLS feature prevents the database server from returning the specified range of bytes literally when this range contains partial characters. If your database locale supports a multibyte code set and you specify a particular column substring in a query, the database server replaces any truncated multibyte characters with single-byte white spaces.

For example, suppose the **multi_col** column contains the string $A^1A^2A^3A^4B^1B^2B^3B^4$ and you execute the following SELECT statement:

```
SELECT multi_col FROM tablename WHERE multi_col[2,4] = 'A1A2B1B2'
```

The query indicates that no matching rows were found because the database server converts the substring **multi_col[2,4]**, the string $A^2A^3A^4$, to three single-byte spaces (sss). The WHERE clause of the query specifies the following condition for the search:

```
WHERE 'sss' = 'A1A2A3'
```

Because this condition is never true, the query retrieves no matching rows.

Informix database servers replace partial characters in each individual substring operation, even when they are concatenated. For example, suppose the **multi_col** column contains $A^1A^2B^1B^2C^1C^2D^1D^2$, and the WHERE clause contains the following condition:

```
multi_col[2,4] | multi_col[6,8]
```

The query does not return any rows because the result of the concatenation ($A^2B^1B^2C^2D^1D^2$) contains two partial characters, A^2 and C^2 . The Informix database server converts these partial characters to single-byte spaces, and creates the following WHERE clause condition:

```
WHERE 'sB1B2sD1D2' = 'A1A2B1B2'
```

This condition is also never true, so the query retrieves no matching rows.

Misinterpreting Partial Characters

Partial characters present a problem if the substrings strings can be processed or presented to users in any way that makes their concatenation not reconstruct the original logical string. Possible problem areas include when a substring of one multibyte character is actually a valid character by itself. For example, suppose a multibyte code set contains a 4-byte character, $A^1A^2A^3A^4$, that represents the digit 1 and a 3-byte character, $A^2A^3A^4$, that represents the digit 6. Suppose also that you use the locale that contains this multibyte code set when you execute the following query:

```
SELECT multi_col FROM tablename WHERE multi_col[2,4] = 'A2A3A4'
```

The database server interprets **multi_col[2,4]** as the valid 3-byte character (a multibyte 6) instead of a substring of the valid 4-byte character ('sss'). Therefore, the WHERE clause contains the following condition:

```
WHERE '6' = '6'
```

The problem of partial characters does not occur in single-byte code sets because each character is stored in a single byte. When your database locale supports a single-byte code set, and you specify a particular column substring in a query, the database server returns exactly the subset of data that you requested and does not replace any characters with white spaces.

Partial Characters in an ORDER BY Clause

Partial characters might also create a problem when you specify column substrings in an ORDER BY clause of a SELECT statement. The general format for specifying column substrings in the ORDER BY clause is as follows:

```
ORDER BY column-name [first, last]
```

In this format, *first* represents the first byte of the first character in the column substring, and *last* represents the last byte of the last character in the column substring.

If the locale supports a multibyte code set whose characters are all of the same length, you can use column substrings in an ORDER BY clause. However, the more likely scenario is that your multibyte code set contains characters with varying lengths. In this case, you might not find it useful to specify column substrings in the ORDER BY clause.

For example, suppose that you want to retrieve all rows from the **multi_data** table, and that you want to use the **multi_chars** column with a column subscript to collate the query results. The following SELECT statement attempts to collate the query results according to the portion of the **multi_chars** column that is contained in the fourth to sixth characters of the column:

```
SELECT * FROM multi_data  
ORDER BY multi_chars[7,12]
```

If the locale supports a multibyte code set whose characters are all 2 bytes in length, then you know that the fourth character in the column begins in byte position 7, and the sixth character in the column ends in byte position 12. The preceding SELECT statement does not generate partial characters.

However, if your multibyte code set contains a mixture of single-byte characters, 2-byte characters, and 3-byte characters, the column substring **multi_chars[7,12]** might create partial characters from the **multi_chars** data. In this case, you might get unexpected results when you specify a column substring in the ORDER BY clause.

For information on the collation order of different types of character data in the ORDER BY clause, see [“The ORDER BY Clause” on page 3-21](#). For the complete syntax and usage of the ORDER BY clause, see the SELECT statement in the [Informix Guide to SQL: Syntax](#).



***Tip:** A partial character might also be generated when an ESQL program copies multibyte data from one buffer to another. For more information, see [“Avoiding Partial Characters” on page 7-25](#).*

Avoidance in BYTE and TEXT Columns

Partial characters are not a problem when you specify a column substring for a column with the BYTE or TEXT data type. The database server avoids partial characters in BYTE and TEXT columns in the following way:

- Because the database server interprets a BYTE column as a series of bytes, not characters, the splitting of multibyte characters as a result of the byte range that a column substring specifies is not an issue.

A column substring for a BYTE column returns the exact range of bytes that is specified and does not replace any bytes with white spaces.

- The database server interprets a TEXT column as a series of characters.

A column substring for a TEXT column returns the exact range of bytes that is specified. Attempts to resolve partial characters in TEXT data are resource intensive. Therefore, the database server does not replace any bytes with white spaces. For more information on the TEXT data type, see [page 3-19](#).



***Warning:** Because your application handles the interpretation of BYTE and TEXT data, it must handle possible occurrences of partial characters in multibyte blob data.*

Specifying Arguments to the TRIM Function

The TRIM function is an SQL function that removes leading or trailing pad characters from a character string. By default, this pad character is an ASCII space. If your locale supports a code set that defines a different character as a space, TRIM does not remove this locale-specific space from the front or back of a string. If you specify the LEADING, TRAILING, or BOTH keywords for TRIM, you can define a different pad character. However, you cannot specify a non-ASCII character as a pad character, even if your locale supports a code set that defines the non-ASCII character.

Using SQL Length Functions

You can use SQL length functions in the SELECT statement and other data manipulation statements. Length functions return the length of a column, string, or variable in bytes or characters.

The choice of locale affects the following three SQL length functions:

- The LENGTH function
- The OCTET_LENGTH function
- The CHAR_LENGTH (or CHARACTER_LENGTH) function

For the complete syntax of these functions, see the Expression segment in the [Informix Guide to SQL: Syntax](#).

The LENGTH Function

The LENGTH function returns the number of bytes of data in character data. However, the behavior of the LENGTH function varies with the type of argument that the user specifies. The argument can be a quoted string, a character-type column other than the TEXT data type, a TEXT column, a host variable, or a stored-procedure variable.

The following table shows how the LENGTH function operates on each of these argument types. The Example column in this table uses the symbol *s* to represent a single-byte trailing white space. This table also assumes that the sample strings consist of single-byte characters.

LENGTH Argument	Behavior	Example
Quoted string	Returns number of bytes in string, minus any trailing white spaces as defined in the locale.	If the string is 'Ludwig', the result is 6. If the string is 'Ludwigsss', the result is still 6.
CHAR, VARCHAR, NCHAR, or NVARCHAR column	Returns number of bytes in a column, minus any trailing white spaces, regardless of defined length of column.	If the fname column of the customer table is a CHAR(15) column, and this column contains the string 'Ludwig', the result is 6. If the fname column contains the string 'Ludwigsss', the result is still 6.
TEXT column	Returns number of bytes in a column, including trailing white spaces.	If the cat_descr column in the catalog table is a TEXT column, and this column contains the string 'Ludwig', the result is 6. If the cat_descr column contains the string 'Ludwigsss', the result is 10.
Host or procedure variable	Returns number of bytes contained in the variable, minus any trailing white spaces, regardless of defined length of the variable.	If the procedure variable f_name is defined as CHAR(15), and this variable contains the string 'Ludwig', the result is 6. If the f_name variable contains the string 'Ludwigsss', the result is still 6.

With Single-Byte Code Sets

When you use the default locale or any locale with a single-byte code set, the LENGTH function seems to return the number of characters in the column. In the following example, the **stores7** database, which contains the **customer** table, uses the default code set for the U.S. English locale. Suppose a user enters a SELECT statement with the LENGTH function to display the last name, length of the last name, and customer number for rows where the customer number is less than 106.

```
SELECT lname AS cust_name, length (fname) AS length, customer_num AS cust_num
FROM customer
WHERE customer_num < 106
```

The following sample of output shows the result of the query. For each row that is retrieved, the **length** column seems to show the number of characters in the **lname** (**cust_name**) column. However, the **length** column actually displays the number of bytes in the **lname** column. In the default code set, 1 byte stores 1 character. (For more information about the default code set, see [“The Default Locale” on page 1-18.](#))

cust_name	length	cust_num
Ludwig	6	101
Carole	6	102
Philip	6	103
Anthony	7	104
Raymond	7	105

With Multibyte Code Sets

When you use the LENGTH function in a locale that supports a multibyte code set, such as the Japanese SJIS code set, the distinction between characters and bytes is meaningful. The LENGTH function returns the number of bytes in the column or quoted string, and this result might be quite different from the number of characters in the string.

The following example assumes that the database that contains the **customer_multi** table has a database locale that supports a multibyte code set. Suppose that the user enters a SELECT statement with the LENGTH function to display the last name, the length of the last name, and the customer number for the customer whose customer number is 199.

```
SELECT lname AS cust_name, length (fname) AS length, customer_num AS cust_num
FROM customer_multi
WHERE customer_num = 199
```

Assume that the last name (**lname**) for customer 199 consists of four characters, represented as follows:

aA¹A²bB¹B²

In this representation, the first character (the symbol a) is a single-byte character. The second character (the symbol A¹A²) is a 2-byte character. The third character (the symbol b) is a single-byte character. The fourth character (the symbol B¹B²) is a 2-byte character.

The following sample of output shows the result of the query. Although the customer first name consists of 4 characters, the length column shows that the total number of bytes in this name is 6.

cust_name	length	cust_num
aA ¹ A ² bB ¹ B ²	6	199

The OCTET_LENGTH Function

The OCTET_LENGTH function returns the number of bytes and generally includes trailing white spaces in the byte count. This SQL length function uses the definition of white space that the locale defines. OCTET_LENGTH returns the number of bytes in a character column, quoted string, host variable, or procedure variable. However, the actual behavior of the OCTET_LENGTH function varies with the type of argument that the user specifies.

The following table shows how the OCTET_LENGTH function operates on each of the argument types. The Example column in this table uses the symbol *s* to represent a single-byte trailing white space. For simplicity, the Example column also assumes that the sample strings consist of single-byte characters.

OCTET_LENGTH Argument	Behavior	Example
Quoted string	Returns number of bytes in string, including any trailing white spaces.	If the string is 'Ludwig', the result is 6. If the string is 'Ludwigssss', the result is 10.
CHAR or NCHAR column	Returns number of bytes in string, including trailing white spaces. This value is the defined length, in bytes, of the column.	If the fname column of the customer table is a CHAR(15) column, and this column contains the string 'Ludwig', the result is 15. If the fname column contains the string 'Ludwigsss', the result is still 15.
VARCHAR or NVARCHAR column	Returns number of bytes in string, including trailing white spaces. This value is the actual length, in bytes, of the character string. It is not the defined maximum column size.	If the cat_advert column of the catalog table is a VARCHAR(255, 65) column, and this column contains the string "Ludwig", the result is 6. If the column contains the string 'Ludwigssss', the result is 10.
TEXT column	Returns number of bytes in column, including trailing white spaces.	If the cat_descr column in the catalog table is a TEXT column, and this column contains the string 'Ludwig', the result is 6. If the cat_descr column contains the string 'Ludwigssss', the result is 10.
Host or procedure variable	Returns number of bytes that the variable contains, including any trailing white spaces, regardless of defined length of variable.	If the procedure variable f_name is defined as CHAR(15), and this variable contains the string 'Ludwig', the result is 6. If the f_name variable contains the string 'Ludwigssss', the result is 10.

The difference between the LENGTH and OCTET_LENGTH functions is that OCTET_LENGTH generally includes trailing white spaces in the byte count, whereas LENGTH generally excludes trailing white spaces from the byte count.

The advantage of the OCTET_LENGTH function over the LENGTH function is that the OCTET_LENGTH function provides the actual column size whereas the LENGTH function trims the column values and returns the length of the trimmed string. This advantage of the OCTET_LENGTH function applies both to single-byte code sets such as ISO8859-1 and multibyte code sets such as the Japanese SJIS code set.

The following table shows some results that the OCTET_LENGTH function might generate.

OCTET_LENGTH Input String	Description	Result
'abc '	A quoted string with 4 single-byte characters (the characters <code>abc</code> and 1 trailing space)	4
'A ¹ A ² B ¹ B ² '	A quoted string with 2 multibyte characters	4
'aA ¹ A ² bB ¹ B ² '	A quoted string with 2 single-byte and 2 multibyte characters	6

The CHAR_LENGTH Function

The CHAR_LENGTH function (also known as the CHARACTER_LENGTH function) returns the number of characters in a quoted string, column with a character data type, host variable, or procedure variable. However, the actual behavior of this function varies with the type of argument that the user specifies.

The following table shows how the CHAR_LENGTH function operates on each of the argument types. The Example column in this table uses the symbol *s* to represent a single-byte trailing white space. For simplicity, the Example column also assumes that the sample strings consist of single-byte characters.

CHAR_LENGTH Argument	Behavior	Example
Quoted string	Returns number of characters in string, including any trailing white spaces as defined in the locale.	If the string is 'Ludwig', the result is 6. If the string is 'Ludwigssss', the result is 10.
CHAR or NCHAR column	Returns number of characters in string, including trailing white spaces. This value is the defined length, in bytes, of the column.	If the fname column of the customer table is a CHAR(15) column, and this column contains the string 'Ludwig', the result is 15. If the fname column contains the string 'Ludwigssss', the result is 15.
VARCHAR or NVARCHAR column	Returns number of characters in string, including white spaces. This value is the actual length, in bytes, of the character string. It is not the defined maximum column size.	If the cat_advert column of the catalog table is a VARCHAR(255, 65) and this column contains the string "Ludwig", the result is 6. If the column contains the string 'Ludwigssss', the result is 10.
TEXT column	Returns number of characters in column, including trailing white spaces.	If the cat_descr column in the catalog table is a TEXT column, and this column contains the string 'Ludwig', the result is 6. If the cat_descr column contains the string 'Ludwigssss', the result is 10.
Host or procedure variable	Returns number of characters that the variable contains, including any trailing white spaces, regardless of defined length of variable.	If the procedure variable f_name is defined as CHAR(15), and this variable contains the string 'Ludwig', the result is 6. If the f_name variable contains the string 'Ludwigssss', the result is 10.

The CHAR_LENGTH function is especially useful with multibyte code sets. If a quoted string of characters contains any multibyte characters, the number of characters in the string differs from the number of bytes in the string. You can use the CHAR_LENGTH function to determine the number of characters in the quoted string.

However, the CHAR_LENGTH function can also be useful in single-byte code sets. In these code sets, the number of bytes in a column is equal to the number of characters in the column. If you use the LENGTH function to determine the number of bytes in a column (which is equal to the number of characters in this case), LENGTH trims the column values and returns the length of the trimmed string. In contrast, CHAR_LENGTH does not trim the column values but returns the actual size of the column.

The following table shows some results that the CHAR_LENGTH function might generate for quoted strings.

CHAR_LENGTH Input String	Description	Result
'abc '	A quoted string with 4 single-byte characters (the characters abc and 1 trailing space)	4
'A ¹ A ² B ¹ B ² '	A quoted string with 2 multibyte characters	2
'aA ¹ A ² bB ¹ B ² '	A quoted string with 2 single-byte and 2 multibyte characters	4

Handling MONEY Columns

The MONEY data type stores currency amounts. This data type stores fixed-point decimal numbers up to a maximum of 32 significant digits. You can specify MONEY columns in data definition statements such as CREATE TABLE and ALTER TABLE.

The choice of locale affects monetary data in the following ways:

- The value of the default scale parameter in the definition of MONEY columns
- The currency notation that the client application uses

The locale defines the default scale and currency notation in the MONETARY category of the locale file. For information on the MONETARY category of the locale file, see [“The MONETARY Category” on page A-7](#).

Default Values for the Scale Parameter

Define a MONEY column with the following syntax:

```
MONEY(precision, scale)
```

Internally, the database server stores MONEY values as DECIMAL values. The *precision* parameter defines the total number of significant digits, and the *scale* parameter defines the total number of digits to the right of the decimal separator. For example, if you define a column as MONEY(8,3), the column can contain a maximum of eight digits, and three of these digits are to the right of the decimal separator. A sample data value in the column might be 12345.678.

If you omit the *scale* parameter from the declaration of a MONEY column, the database server provides a scale that the locale defines. For the default locale (U.S. English), the database server uses a default scale of 2. It stores the data type MONEY(*precision*) in the same internal format as the data type DECIMAL(*precision*,2). For example, if you define a column as MONEY(10), the database server creates a column with the same format as the data type DECIMAL(10,2). A sample data value in the column might be 12345678.90.

For nondefault locales, if you omit the scale when you declare a MONEY column, the database server declares a column with the same internal format as DECIMAL data types with a locale-specific default scale. For example, if you define a column as MONEY(10), and the locale defines the default scale as 4, the database server stores the data type of the column in the same format as DECIMAL(10,4). A sample data value in the column might be 123456.7890.

For the complete syntax of the MONEY data type, see the [Informix Guide to SQL: Syntax](#). For a complete description of the MONEY data type, see the [Informix Guide to SQL: Reference](#).

Format of Currency Notation

Client applications format values in MONEY columns with the currency notation that the locale defines. This notation specifies the currency symbol, thousands separator, and decimal separator. (For more information on currency notation, see [“Numeric and Monetary Formats” on page 1-14.](#))

For the default locale, the currency symbol is a dollar sign (\$), the thousands separator is a comma (,), and the decimal separator is a period (.). For nondefault locales, the locale defines the appropriate culture-specific currency notation for monetary values. You can also use the **DBMONEY** environment variable to customize the currency symbol and decimal separator for monetary values. For more information, see [“Customizing Monetary Values” on page 1-53.](#)

Using Data Manipulation Statements

The choice of a locale can affect the following SQL data manipulation statements:

- DELETE
- INSERT
- LOAD
- UNLOAD
- UPDATE

The following sections describe the GLS aspects of these SQL statements. For a complete description of the use and syntax of these statements, see Chapter 1 of the [Informix Guide to SQL: Syntax](#).

Specifying Conditions in the WHERE Clause

The following SQL statements might include a WHERE clause to tell the database server on which rows to operate:

- For the DELETE statement, the WHERE clause specifies which rows to delete.
- For the INSERT statement if the statement includes an embedded SELECT, the WHERE clause specifies which rows to insert from another table.
- For the UPDATE statement, the WHERE clause specifies which rows to update. In addition, the SET clause can include an embedded SELECT statement whose WHERE clause identifies a row whose values are to be assigned to another row.
- For the UNLOAD statement, the WHERE clause of the embedded SELECT specifies which rows to unload.

The choice of a locale affects these uses of a WHERE clause in the same way that it affects the WHERE clause of a SELECT. For more information, see [“Logical Predicates in a WHERE Clause” on page 3-24](#) and [“Comparisons with MATCHES and LIKE Conditions” on page 3-28](#).

Specifying Era-Based Dates

The following SQL statements might specify DATE and DATETIME column values:

- The WHERE clause of the DELETE statement
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement

When you specify a DATE column value in one of the preceding SQL statements, the database server uses the **GL_DATE** (or **DBDATE**) environment variable to interpret the date expression, as follows:

- If you have set **GL_DATE** (or **DBDATE**) to an era-based (Asian) date format, you can use era-based date formats for date expressions.
- If you have not set the **GL_DATE** (or **DBDATE**) environment variable to an era-based date format, you can use era-based date formats for date expressions *only* if the server processing locale supports era-based dates. For more information on the server processing locale, see [“Determining the Server-Processing Locale” on page 1-34](#).
- If your locale does not support era-dated dates, you cannot use era-based date formats for date expressions. If you attempt to specify an era-based date format in this case, the SQL statement fails.

When you specify a DATETIME column value, the database server uses the **GL_DATETIME** (or **DBTIME**) environment variable instead of the **GL_DATE** (or **DBDATE**) environment variable to interpret the expression.

For more information, see [“Era-Based Date and Time Formats” on page 1-51](#). For information on how to set the **GL_DATE**, **GL_DATETIME**, **DBDATE**, and **DBTIME** environment variables to an era-based format, see their entries in [Chapter 2, “GLS Environment Variables.”](#)

Loading and Unloading Data

The LOAD and UNLOAD statements allow you transfer data to and from your database with operating-system text files. The following sections describe the GLS aspects of the LOAD and UNLOAD statements. For a complete description of the use and syntax of these statements, see Chapter 1 of the [Informix Guide to SQL: Syntax](#).

Loading Data into a Database

The LOAD statement inserts data from an operating-system text file into a table or view. This operating-system text file is called a LOAD FROM file. The data in this file can contain any character that the client code set defines. If the client locale supports a multibyte code set, this data can contain multibyte characters. If the database locale supports a code set that is different from but convertible to the client code set, the client performs code-set conversion on the data before it sends this data to the database server. For more information, see [“Performing Code-Set Conversion” on page 1-40](#).

The locale also defines the formats for date, time, numeric, and monetary data. You can use any format that the client locale supports as a column value in the LOAD FROM file. For example, a French locale might define monetary values that have a space as the thousands separator and a comma as the decimal separator. When you use this locale, the following MONEY column value is valid in a LOAD FROM file:

```
3 411,99
```

You can set environment variables to specify alternative end-user formats for date and monetary data. If you set these environment variables, the LOAD FROM files can use the alternative end-user formats for DATE, DATETIME, and MONEY column values. For more information, see [“Customizing Date and Time End-User Formats” on page 1-50](#) and [“Customizing Monetary Values” on page 1-53](#).

Unloading Data from a Database

The UNLOAD statement writes the rows that a SELECT statement retrieves to an operating-system text file. This operating-system text file is called an UNLOAD TO file. The data in this file contains characters that the client code set defines. If the client locale supports a multibyte code set, this data can contain the multibyte characters. If the database locale supports a code set that is different from but convertible to the client code set, the client performs code-set conversion on the data before it writes this data to the UNLOAD TO file. (For more information, see [“Performing Code-Set Conversion” on page 1-40](#).)



The client locale and certain environment variables determine the output format of certain data types in the UNLOAD TO file. These data types include DATE values, MONEY values, values of numeric data types, and DATETIME values. For further information, see [“End-User Formats” on page 1-12](#) and [“How Do You Customize Client End-User Formats?” on page 1-49](#).

Important: You can use an UNLOAD TO file, which the UNLOAD statement generates, as the input file (the LOAD FROM file) to a LOAD statement that loads another table or database. When you use an UNLOAD TO file in this manner, make sure that all environment variables and the client locale have the same values when you perform the LOAD as they did when you performed the UNLOAD.

INFORMIX-Universal Server Features

Using the Server Locale	4-3
Generating Non-ASCII Filenames	4-4
The GLS8BITFSYS Environment Variable	4-5
The Server Code Set	4-6
Performing Code-Set Conversion	4-6
Which Data Does Universal Server Convert?	4-7
Support for Locales by Universal Server Utilities	4-8
Locale Environment Variables.	4-9
Non-ASCII Characters in Command-Line Arguments	4-10
Universal Server Data Types	4-11
SQL-92 Data Types	4-12
LVARCHAR Data Type	4-13
Smart Large Objects	4-13
Complex Data Types	4-14
Opaque Data Types	4-14
Input and Output Support Functions	4-15
Send and Receive Support Functions	4-16
Distinct Data Types	4-16
SQL Identifiers That Support Non-ASCII Characters	4-17
Introducing the Informix GLS API	4-18
Compliance with Industry Standards	4-18
How to Use the GLS API	4-19
Compiling and Linking the GLS API	4-19

How to Internationalize Programs with the GLS API	4-20
String Traversal	4-21
String Processing	4-22
Code-Set Conversion	4-24
Character Classification.	4-25
Case Conversion	4-26
Character/String Comparison and Sorting	4-27
Date/Time Conversion and Formatting Functions	4-27
Numeric Conversion and Formatting	4-28
Money Conversion and Formatting	4-28
Stream Input and Output	4-29
Initialization and Error Handling	4-29
Accessing Messages	4-30
Improving the Performance of Your Programs	4-30

This chapter explains how the GLS feature affects INFORMIX-Universal Server. It contains the following topics:

- When Informix uses the server locale
- How Informix utilities provide support for the GLS feature
- How the new data types might affect Global Language Support
- The new GLS Application Programming Interface (API)

For descriptions of Universal Server features that are not unique to the GLS feature, refer to the [INFORMIX-Universal Server Administrator's Guide](#). For an introduction to the GLS feature, see [Chapter 1, "GLS Fundamentals."](#)

Using the Server Locale

Universal Server might perform read and write operations to the following operating-system files:

- Diagnostic files
Universal Server generates diagnostic files when you set one or more of the following configuration parameters: DUMPCNT, DUMPCORE, DUMPDIR, DUMPGCORE, and DUMPSHMEM. Diagnostic files include the **af.***, **shmem.xxx**, **gcore.xxx**, and **core** files. For more information on these files, refer to the [INFORMIX-Universal Server Administrator's Guide](#).
- Message-log file
Universal Server generates a user-specified message-log file when you set the MSGPATH configuration parameter.

***Tip:** For more information on diagnostic files, message logs, and configuration parameters, see the "[INFORMIX-Universal Server Administrator's Guide](#)."*



These operating-system files reside on the server computer, the same computer on which Universal Server resides. When Universal Server reads from or writes to these files, it must use a code set that the server computer supports. Universal Server obtains this code set from the server locale. You set the server locale with the **SERVER_LOCALE** environment variable. If you do not set **SERVER_LOCALE**, Universal Server uses the default locale, U.S. English, as the server locale. For the definition of server locale, see [page 1-27](#). For information on how to set **SERVER_LOCALE**, see [page 2-41](#).

Universal Server needs to determine the code set that the server locale supports (the server code set) to perform the following tasks that are associated with the read/write operations to its operating-system files:

- Generate filenames that contain non-ASCII characters
- Perform code-set conversion between the code sets of the server-processing locale and the server locale

Generating Non-ASCII Filenames

You can use non-ASCII characters (8-bit and multibyte characters) when you create or refer to any of the following Universal Server names:

- Chunk name
- Message-log filename
- A pathname

The following restrictions affect the ability of Universal Server to generate filenames that contain non-ASCII characters:

- Universal Server must know whether the operating system is 8-bit clean.
- Your server code set must support these non-ASCII characters.

The GLS8BITFSYS Environment Variable

The **GLS8BITFSYS** environment variable on the server computer tells Universal Server whether the operating system is 8-bit clean. Universal Server uses the value of **GLS8BITFSYS** to determine whether it can use non-ASCII characters in the filename of an operating-system file that it generates, as follows:

- When **GLS8BITFSYS** is 1, Universal Server can use non-ASCII characters in filenames of its operating-system files.
- When **GLS8BITFSYS** is 0, Universal Server does *not* allow non-ASCII characters in filenames of its operating-system files.

For example, the MSGPATH configuration parameter specifies a filename for a Universal Server message log. Suppose you specify a Universal Server message log that is called $/A^1A^2B^1B^2/C^1C^2D^1D^2$, where A^1A^2 , B^1B^2 , C^1C^2 , and D^1D^2 are multibyte characters, with the configuration parameter in your ONCONFIG configuration file that Figure 4-1 shows.

```
MSGPATH /A1A2B1B2/C1C2D1D2  # multibyte message-log filename
```

Figure 4-1
*Message-Log
Filename with
Multibyte
Characters*

If **GLS8BITFSYS** is 1 (or is not set) on the server computer, Universal Server assumes that the operating system is 8-bit clean, and it generates a file that is called $C^1C^2D^1D^2$ in the $/A^1A^2B^1B^2$ directory. For more information on an 8-bit clean operating system and the **GLS8BITFSYS** environment variable, see [page 2-23](#).

The Server Code Set

When Universal Server actually creates a file whose filename has non-ASCII characters, the server locale must support these non-ASCII characters. Before you start Universal Server, you must set the **SERVER_LOCALE** environment variable to the name of a locale whose code set contains these non-ASCII characters.

For example, suppose you want a Universal Server message log with the path $/A^1A^2B^1B^2/C^1C^2D^1D^2$, where A^1A^2 , B^1B^2 , C^1C^2 , and D^1D^2 are multibyte characters in the Japanese SJIS code set. You must perform the following steps to enable Universal Server to create this message-log file on the server computer:

1. Modify the MSGPATH configuration parameter in the ONCONFIG file, as the example in [Figure 4-1 on page 4-5](#) shows.
2. Set the **SERVER_LOCALE** environment variable on the server computer to the Japanese SJIS locale, **ja_jp.sjis**.
3. Start Universal Server with the **oninit** utility.

When Universal Server initializes, it creates the $/A^1A^2B^1B^2/C^1C^2D^1D^2$ message log. For more information on **oninit**, see the [INFORMIX-Universal Server Administrator's Guide](#).

Performing Code-Set Conversion

Once Universal Server creates the operating-system file, it has generated a filename and written file contents in the code set of the server locale (the server code set). Any Informix product or client application that needs to access this file must have a server-processing locale that supports this same server code set. You must ensure that the appropriate locale environment variables are set so that the server-processing locale supports a code set with these non-ASCII characters. (For more information on the server-processing locale, see [page 1-34](#).)

Universal Server checks the validity of a filename with respect to the server-processing locale before it references a filename. When Universal Server transfers data to and from its operating-system files, it handles any differences in the code sets of the server-processing locale and the server locale, as follows:

- If these two code sets are the same, Universal Server can read from or write to its operating-system files in the code set of the server locale.
- If these two code sets are different, and an Informix code-set conversion exists between them, Universal Server automatically performs code-set conversion when it reads from or writes to its operating-system files.

For code-set conversion to resolve the difference in code sets, the server locale must support the actual code set that Universal Server used to create the file. For more information, see [“The Server Code Set” on page 4-6](#).

- If these two code sets are different, but no Informix code-set conversion exists, Universal Server cannot perform code-set conversion.

If Universal Server reads from or writes to an operating-system file for which no code-set conversion exists, it uses the code set of the server-processing locale to perform the read or write operation.

Which Data Does Universal Server Convert?

When the code sets of two locales differ, an Informix client product must use code-set conversion to prevent data corruption of character data. Code-set conversion converts the following types of character data:

- SQL data types
 - CHAR and VARCHAR
 - NCHAR and NVARCHAR
 - LVARCHAR
 - TEXT (the BYTE data type is *not* converted)
- Any of the ESQL/C character data types (**char**, **fixchar**, **string**, and **varchar**)



- SQL statements, both static and dynamic
- Database objects such as:

- Column names
- Table names
- Statement-identifier names
- Cursor names

For a complete list of SQL identifiers, see [“SQL Identifiers That Support Non-ASCII Characters”](#) on page 4-17.

- Stored procedure text
- Command text
- Error message text in the `sqlca.sqlerrm` field

***Tip:** If your ESQL/C client application uses code-set conversion, you might need to take special programming steps. For more information, see [“Handling Code-Set Conversion”](#) on page 8-12.*

For more information on code-set conversion, see [page 1-40](#).

Support for Locales by Universal Server Utilities

The GLS feature enables Universal Server utilities to provide the following locale-specific support:

- Locale environment variables to establish nondefault locales
- Non-ASCII characters in command-line arguments

This section provides information that is specific to the use of the GLS feature by Universal Server utilities. For a complete description of Universal Server or OnLine Dynamic Server utilities, refer to the [INFORMIX-Universal Server Administrator's Guide](#) or the *INFORMIX-OnLine Dynamic Server Administrator's Guide*. For a description of SE utilities, refer to the [Informix Guide to SQL: Reference](#).

Locale Environment Variables

Universal Server utilities and SQL utilities are client applications that request information from an instance of Universal Server. Therefore, these utilities use the **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables to obtain the name of a nondefault locale, as follows:

- If a database utility is to use a nondefault code set to accept input, (including command-line arguments) and to generate output, you must set the **CLIENT_LOCALE** environment variable.
- If a database utility accesses a database with a nondefault locale, you must set the **DB_LOCALE** environment variable.
- If a database utility causes Universal Server to write data on the server computer that has a nondefault code set, you must set the **SERVER_LOCALE** environment variable.

For more information on these environment variables, see [Chapter 2, “GLS Environment Variables.”](#)

These utilities also perform code-set conversion if the database and the client locales support convertible code sets. For more information on code-set conversion, see [page 1-40](#).

Non-ASCII Characters in Command-Line Arguments

Most Universal Server utilities support non-ASCII characters in command-line arguments. These utilities interpret all command-line arguments in the client code set (which `CLIENT_LOCALE` defines). Figure 4-2 shows Universal Server utilities that accept non-ASCII characters in command-line arguments.

Figure 4-2
Universal Server Utilities That Accept Non-ASCII Characters in Command-Line Arguments

Utility Name	Non-ASCII Characters in Command-Line Argument	Non-ASCII Output
onaudit	<code>-f input_file</code>	Yes
oncheck	<code>-cc -pc database</code> <code>-ci -cI -pk -pK -pl -pL database:table#index_name</code> <code>-ci -cI -pk -pK -pl -pL -cd -cD -pB -pt -pT -pd -pD -pp database:table</code>	Yes
onload	<code>database:table</code> <code>-i old_index new_index</code> <code>-t tape_device</code>	Yes
onlog	<code>-d tape_device</code>	None
onshowaudit	<code>-f input_file</code> <code>-s server_name</code>	Yes
onspaces	<code>-p pathname</code> <code>-f filename</code>	None
onstat	<code>-o filename -dest filename_source</code>	Yes
onunload	<code>database:table</code> <code>-t tape_device</code>	Yes

The DB-Access utility generates labels and messages in the code set of the client locale. ♦

The following SQL utilities also accept non-ASCII characters in command-line arguments and generate any output in the client code set:

- **chkenv**
- **dbexport**
- **dbimport**
- **dbload**
- **dbschema**

For a description of these SQL utilities, refer to the [Informix Guide to SQL: Reference](#). For information about how to use these utilities to migrate to or from a database, see the [Informix Migration Guide](#).

Universal Server Data Types

Universal Server provides support for the following data types:

- Traditional SQL-92 data types
- LVARCHAR data type
- Smart large objects
- Complex data types: collections and row types
- Opaque data types
- Distinct data types

In addition, Universal Server provides the following built-in data types:

- Simple large objects: TEXT and BYTE
- BOOLEAN

TEXT, BYTE, and BOOLEAN data types have no GLS considerations.

The following sections summarize the GLS considerations for the SQL-92, LVARCHAR, smart-large-object, complex, opaque, and distinct data types.

SQL-92 Data Types

Universal Server supports the traditional SQL-92 data types, as the following table shows.

Traditional SQL-92 Data Type	INFORMIX-Universal Server Implementation
integer	SMALLINT, INTEGER, INT8, SERIAL, SERIAL8
floating-point number	SMALLFLOAT, FLOAT
character string:	
fixed	CHAR, NCHAR
varying	VARCHAR, NVARCHAR
date and time (including interval)	DATE, DATETIME, INTERVAL
numeric and decimal	DECIMAL, MONEY

For these data types, Universal Server automatically provides the following locale-related tasks:

- It automatically performs code-set conversion for the character-string data types.
For more information, see the discussion of code-set conversion in [“Performing Code-Set Conversion” on page 1-40](#).
- It automatically uses localized collation order for data in NCHAR and NVARCHAR columns.
For more information on NCHAR and NVARCHAR, see [“Using Character Data Types” on page 3-11](#).
- It automatically provides locale-specific formatting for date and time data and for monetary data.
For more information, see [“End-User Formats” on page 1-12](#).

In addition to the SQL-92 data types, Universal Server provides the following built-in data types.

LVARCHAR Data Type

The LVARCHAR data type is also used to represent the format of a base type value. LVARCHAR is similar to the Informix-specific data type VARCHAR. (VARCHAR stores character data of varying length.)

- The LVARCHAR data type supports values greater than 256 bytes.
- The LVARCHAR data type is collated in code-set collation order.
- Client applications perform code-set conversion on LVARCHAR data.

For more information on code-set collation and conversion, see [Chapter 1, “GLS Fundamentals.”](#)

The LVARCHAR data type supports SQL length functions similar to the VARCHAR data type. For more information, see [“Using SQL Length Functions” on page 3-40.](#)

For more information on the LVARCHAR data type, see Chapter 3 of the *Informix Guide to SQL: Reference*.

Smart Large Objects

A smart large object can store text or images. Smart large objects are stored and retrieved in pieces and have database properties such as crash recovery and transaction rollback. Universal Server supports the following two smart-large-object data types:

- A binary large object (BLOB) stores any type of binary data, including images and video clips.
- A character large object (CLOB) stores text such as PostScript or HTML files.

You can seek smart large objects in bytes, but not in characters. Therefore, you need to manage the byte offset of multibyte characters when you seek information in smart large objects.

Universal Server does *not* perform code-set conversion for either of these smart-large-object data types.

Complex Data Types

Universal Server also provides support for the complex data types:

- Collection types: SET, MULTISSET, and LIST
- Row types: named row types and unnamed row types

Any of these data types can have members with character, date or time, or numeric data types. Universal Server can still handle the GLS concerns for these data types when they are part of a complex data type.

Opaque Data Types

Universal Server provides the capabilities for programmers to define *opaque* data types. An opaque data type is fully encapsulated; its internal structure is not known to the database server. Therefore, the database server cannot automatically perform locale-specific tasks such as code-set conversion for opaque types.

When you create an opaque data type, you must write the support functions and SQL functions of the opaque type so that they handle locale-sensitive data.

Many user-defined functions handle non-ASCII data correctly, even if they were originally written for ASCII data. However, some functions might perform abnormally. Follow these suggested actions to manage these functions:

- If the server locale is not supported by the function, the function can return an error message.
Universal Server provides a new application-programming interface (API) routine to locate the current server locale from within a user function.
- Ensure that the function works correctly with any locale-sensitive issues such as string manipulation or the formatting of date, time, or money values.

For more information, see [“Introducing the Informix GLS API” on page 4-18](#).

When you create an opaque data type, you must write the support functions of the opaque type so that they handle any locale-sensitive data. In particular, consider how to handle any locale-sensitive data when you write the following support functions:

- The input and output support functions
- The receive and send support functions

The following sections summarize GLS considerations for these support functions. For more information on the support functions of an opaque data type, see [Extending INFORMIX-Universal Server: Data Types](#).

Input and Output Support Functions

The internal representation of an opaque data type is the C structure in which it is stored. Each opaque type also has a character-based format, known as its external representation. This external representation is stored in an internal data type called LVARCHAR. The LVARCHAR data type can store single-byte (ASCII and non-ASCII) and multibyte character data. Data of an opaque data type can be transferred to and from the database server in this external representation. Therefore, an opaque data type can hold single-byte or multibyte data.

However, the ability to transfer the data between a client application and database server is not sufficient to support locale-sensitive data. It does not ensure that the data is correctly manipulated at its destination.

The input and output support functions convert the opaque data type from its internal to an external representation, and vice versa. The input function converts the external representation of the data type to the internal representation. The output function converts the internal representation of the data type to the external representation.

These functions should correctly handle any locale-sensitive data. In addition, they should perform any code-set conversion that might be required.

Send and Receive Support Functions

The send and receive functions support binary transfer of opaque data types. That is, they convert the opaque data type from its internal representation on the client computer to its internal representation on the server computer (where it is stored), as follows:

- When a client application sends an opaque type in its internal representation to the database server, the database server invokes the *receive* support function to convert the data from the client internal representation to the server internal representation.
- When a client application receives an opaque type in its internal format from the database server, the database server invokes the *send* support function to convert the data from the server internal representation to the client representation.

If the internal representation of an opaque type contains a string, money, date, or time type, the client application cannot perform any locale translation because the opaque data type is encapsulated. Therefore, the send and receive functions must perform the translation. You can use the following DataBlade API functions to help with these translations:

- **mi_get_string(), mi_put_string()**
- **mi_get_money(), mi_put_money()**
- **mi_get_date(), mi_put_date()**
- **mi_get_datetime(), mi_put_datetime()**

For more information on these DataBlade API functions, see the [*DataBlade API Programmer's Manual*](#).

Distinct Data Types

A distinct data type has the same internal storage representation as its source type but has a different name. Its source type can be an existing opaque data type, built-in data type, named row type, or another distinct data type. Universal Server handles GLS considerations for a distinct type as it would the source type.

SQL Identifiers That Support Non-ASCII Characters

If you use a nondefault locale that supports a code set with non-ASCII characters, you can use these non-ASCII characters to form most SQL identifiers. Universal Server can support non-ASCII characters in the following SQL identifiers:

- The SQL identifiers that *all* Informix database servers support
[Figure 3-1 on page 3-5](#) lists the SQL identifiers in which *all* Informix database servers can support non-ASCII characters.
- Some of the SQL identifiers that are unique to Universal Server
[Figure 4-3](#) shows the additional SQL identifiers in which Universal Server can support non-ASCII characters. The SQL Identifier column lists the name of the database object. The SQL Segment column shows the segment that gives the complete syntax of the identifier in the *Informix Guide to SQL: Syntax*. The Notes column describes any special considerations for the identifier and also provides an example of an SQL statement that creates or uses the identifier.

Figure 4-3

SQL Identifiers Unique to INFORMIX-Universal Server That Support Non-ASCII Characters

SQL Identifier	SQL Segment	Notes
Cast	Expression	An example of an SQL statement that uses a cast name is CREATE CAST.
Distinct data type	Data Type	An example of an SQL statement that uses a distinct-data-type name is CREATE DISTINCT.
Function	Function Name	An example of an SQL statement that uses a function name is CREATE FUNCTION.
Opaque data type	Data Type	An example of an SQL statement that uses a opaque-data-type name is CREATE OPAQUE TYPE.

(1 of 2)

SQL Identifier	SQL Segment	Notes
Operator class	Expression	An example of an SQL statement that uses a operator-class name is CREATE OPCLASS.
Procedure	Procedure Name	An example of an SQL statement that uses a procedure name is CREATE PROCEDURE.
Row data type	Data Type	An example of an SQL statement that uses a row data type name is CREATE ROW TYPE.

(2 of 2)

Introducing the Informix GLS API

Informix provides the Global Language Support Application-Programming Interface (GLS API) to enable you to develop internationalized applications with a C-language interface. The GLS API relies on GLS locales, which contain culture-specific information. The GLS API provides macros and functions to:

- process single-byte and multibyte characters and strings.
- collate single-byte and multibyte characters and strings.
- process input and output
- convert date, time, money, and number strings from and to binary values.
- convert between code sets.

Compliance with Industry Standards

The GLS API for Universal Server, Version 9.1 was derived from the X/Open XPG4 specifications.

How to Use the GLS API

You can use the GLS API on either the client application or the database server. All functions access the current processing locale. The current processing locale, based on the GLS environment variables and data stored in the database, must be established on both the server computer and client computer before any locale-sensitive processing occurs. The database server establishes the current processing locale when a client application opens a database (or when a session corresponding to a client connection is established). A client application needs to set the processing locale explicitly with a function call.

Compiling and Linking the GLS API

To use the GLS API in your C-language program, you must include the following header file in your source file:

```
#include <ifxgls.h>
```

Both ESQL/C users and client-side users of the DataBlade API need to ensure that the correct processing locale is established by calling the **ifx_gl_init()** function prior to calling any of the **ifx_gl_*** functions.

To compile and link ESQL/C programs that use the GLS API, issue the following ESQL command:

```
% esql <source_file>
```

To compile and link DataBlade module programs, use the following command:

```
% cc -I$INFORMIXDIR/include/esql -L$INFORMIXDIR/lib/esql \ ... -lixgls ... <source_file>
```

Additionally, DataBlade module programmers need to distinguish whether a DataBlade module runs on the server computer or on a client computer when they compile user-defined routines. To make this distinction, use the compiler flag **-DMI_SERVBUILD** when you compile user-defined routines.

Tip: When you use the *DataBlade Developers Kit (DBDK)* to compile user-defined routines, you do not have to explicitly specify the location of the header files.



To build a shared object that contains the C code for a user-defined routine, use the following example:

```
% cc -I$INFORMIXDIR/incl/esql -K pic -c -DMI_SERVBUILD <source_file>
% ld -G -o <shared object name>.so <source_file>.o
```

(For more information on how to write user-defined routines, see the DataBlade Developers Kit.)

The following directories must be available to use the GLS API:

- **\$INFORMIXDIR/gls**
This directory includes the subdirectories for locale and code-set conversion files.
- **\$INFORMIXDIR/lib/esql**
This directory contains the static and shared GLS libraries.
- **\$INFORMIXDIR/incl/esql**
This directory contains two header files: **gls.h** and **ifxgls.h**

How to Internationalize Programs with the GLS API

The ultimate goal of internationalizing an application program is to create (or modify) an application so that the only change necessary to support a different language, territory, and code set is to point the application to the correct GLS locale.

This section discusses the areas you need to consider in your application to perform internationalization. These application areas are listed in order of importance. Each area references a macro or function of the GLS API and includes a brief description. The GLS API is documented on-line as HTML pages with Universal Server. For a list of the macros and functions of the GLS API, refer to the [GLS API Programmer's Manual](#).

The following GLS API macros and functions are for multibyte character processing only. Wide-character macros and functions are listed in [“Improving the Performance of Your Programs”](#) on page 4-30.

String Traversal

String traversal of a multibyte character string can be forward or backward. An example is provided for both directions.

Traversing Multibyte Character Strings Forward

```
gl_mchar_t buf[], *p;
for ( p = buf; *p != '\0' ; p = ifx_gl_mbsnext( p, IFX_GL_NO_LIMIT))
    process_mchar(p);
```

Traversing Multibyte Character Strings Backward

```
gl_mchar_t buf[], *p;
p = ifx_gl_mbsprev(buf, buf + strlen(buf));
if ( p != NULL )
    for ( ; p >= buf; p = ifx_gl_mbsprev(buf, p) )
        process_mchar(p);
```

The null terminator in a multibyte character string is assumed to occupy a single byte. To process a multibyte character, you cannot pass the entire character to a function. You must pass a pointer to the beginning of the character so that the called function can access the remaining bytes of the character.

The GLS API provides the following functions for multibyte string traversal and indexing:

- **ifx_gl_mblen()**
This function determines the number of bytes in the multibyte character, *mb*.
- **ifx_gl_mbsnext()**
This function returns a pointer to the next multibyte character after *mb*.
- **ifx_gl_mbsprev()**
This function returns a pointer to the multibyte character before *mb*, where *mb0* is a pointer to the beginning of the multibyte string.

String Processing

String processing involves concatenating character strings, searching for characters in strings, copying strings (and portions of strings), and so on. The following GLS API functions provide multibyte string processing capabilities:

- **ifx_gl_mbscat()**

This function appends a copy of the character string *mbs2* to the end of the character string *mbs1*. If *mbs1* and *mbs2* overlap, the result of this function is undefined.

- **ifx_gl_mbschr()**

This function locates the first occurrence of *mb* in the multibyte string *mbs*.

- **ifx_gl_mbscpy()**

This function copies the multibyte string *mbs2* to the location pointed to by *mbs1*. If *mbs1* and *mbs2* overlap, the result of this function is undefined.

- **ifx_gl_mbscspn()**

This function returns the number of characters in the maximum initial substring of *mbs1*, which consists entirely of multibyte characters not in the string *mbs2*.

- **ifx_gl_mbslen()**

This function returns the number of characters (not bytes) in the string *mbs*, not including any terminating null characters.

- **ifx_gl_mbsmbs()**

This function searches for the first occurrence of the multibyte string *mbs1* in the multibyte string *mbs2*.

- **ifx_gl_mbsncat()**

This function appends *mbs2* to the end of *mbs1*. No more than *char_limit* characters are read from *mbs2* and written to *mbs1*. If *mbs1* and *mbs2* overlap, the result of this function is undefined.

- **ifx_gl_mbsncpy()**

This function copies *mbs2* to the location to which *mbs1* points. No more than *char_limit* characters are read from *mbs2* and written to *mbs1*. If *mbs1* and *mbs2* overlap, the result of this function is undefined.

- **ifx_gl_mbsntslen()**

This function returns the number of characters in *mbs*, not including the trailing space characters. The characters that are not included in the count are the space characters defined in the current locale.

- **ifx_gl_mbsntsbytes()**

This function returns the number of bytes in *mbs*, not including the trailing space characters. The characters that are not included in the count are the ASCII space characters and any multibyte characters that are equivalent to the ASCII space character.

- **ifx_gl_mbspbrk()**

This function searches for the first occurrence in the multibyte string *mbs1* of any multibyte character from the string *mbs2*.

- **ifx_gl_mbsrchr()**

This function locates the last occurrence of *mb* in the multibyte string *mbs*.

- **ifx_gl_mbsspn()**

This function returns the number of characters in the maximum initial substring of *mbs1*, which consists entirely of multibyte characters in the string *mbs2*.

Memory Allocation

No GLS library function allocates memory that remains after the function returns. If a function allocates memory, this memory is allocated only temporarily and is freed before the function returns. Therefore, the caller of each function must allocate memory needed by the function. The following GLS API macro and function help you determine how much memory a multibyte character requires:

- **IFX_GL_MB_MAX**

This macro indicates the maximum number of bytes that any multibyte character in any locale can occupy. This macro is usually used to allocate space in static buffers that are intended to contain one or more multibyte characters.

- **ifx_gl_mb_loc_max()**

This function returns the maximum number of bytes that any multibyte character can occupy.

Code-Set Conversion

Because a character might be encoded differently on different operating systems, the appropriate communication layer must be prepared to convert between the two encodings. The GLS API provides the following functions to support code-set conversion:

- **ifx_gl_conv_needed()**

This function determines whether characters encoded in *srccharset* require conversion to *dstcharset* by using **ifx_gl_cv_mconv()**. Use this function to determine if code-set conversion is needed. Comparing the names of the code sets does not provide enough information. For example, 8859-1, 819, and Latin-1 all refer to the same code set.

- **ifx_gl_cv_mconv()**

This function converts the string of characters in **src* to other characters but encoded in another code set as defined in the code-set conversion files, *\$INFORMIXDIR/gls/cv**. This function stores the result in the buffer to which **dst* points.

- **ifx_gl_cv_outbuflen()**

This function calculates either exactly the number of bytes that are required by a destination buffer of code-set-converted multibyte characters or a close over-approximation of the number. The argument *srcbytes* is the number of bytes in the buffer of multibyte characters to be code-set converted.

- **ifx_gl_cv_sb2sb_table()**

If the code-set conversion, from *srccharset* to *dstcharset*, converts from a single-byte code set to another single-byte code set where there are no substitution conversions, this function sets *array* to an array of 256 unsigned characters that represent the conversion. If the code-set conversion is not of this form, then this function sets *array* to NULL.

Character Classification

The following GLS API functions test whether the multibyte character for the respective character classification follows the rules of the current locale:

- **ifx_gl_ismalnum()**
This function returns true if the character is either in the alpha or digit class.
- **ifx_gl_ismalpha()**
This function returns true if the character is an alphabetic character. All uppercase and lowercase characters are also in this class.
- **ifx_gl_ismblank()**
This function returns true if the character is a horizontal space character. The single-byte space and tab characters plus any multibyte version of these characters are in this class.
- **ifx_gl_ismcntrl()**
This function returns true if the character is a control character.
- **ifx_gl_ismdigit()**
This function returns true if the character is a digit. Only the 10 ASCII digits are in this class.
- **ifx_gl_ismgraph()**
This function returns true if the character is a graphical character. All characters which have a visual representation are in this class.
- **ifx_gl_ismlower()**
This function returns true if the character is a lowercase alphabetic character.
- **ifx_gl_ismprint()**
This function returns true if the character is a printable character.
- **ifx_gl_ismpunct()**
This function returns true if the character is a punctuation character. A single-byte ASCII punctuation characters plus any non-ASCII punctuation characters are in this class.

- **ifx_gl_ismspace()**

This function returns true if the character is a horizontal or vertical space character. This class includes all characters from the blank class, single-byte and multibyte version of newline, vertical tab, form feed, and carriage return.

- **ifx_gl_ismupper()**

This function returns true if the character is an uppercase alphabetic character. The ASCII characters *A* through *Z* and all other single and multibyte uppercase characters for Latin-based languages are in this class.

- **ifx_gl_ismxdigit()**

This function returns true if the character is a digit character. Only the 10 ASCII digit characters, *A* through *F*, and *a* through *f* are in this class. Multibyte version or alternative representations (for example, Hindi or Kanji digits) are in the alpha class.

Case Conversion

All alphabetic case conversions must use the conversions that the locale specifies. The GLS API provides the following functions to support multibyte case conversion:

- **ifx_gl_case_conv_outbuflen()**

This function calculates either exactly the number of bytes that are required by a buffer of case-equivalent multibyte characters or a close over-approximation of the number.

- **ifx_gl_tomlower(), ifx_gl_tomupper()**

These functions return the alphabetic case equivalent of the source character or return the source character if the source character does not have a case equivalent.

Character/String Comparison and Sorting

All sorting and comparison of characters and character strings must adhere to the comparison order that the locale specifies. The **ifx_gl_mbcoll()** function compares the multibyte character strings *mbs1* and *mbs2* according to the rules of the current locale.

Date/Time Conversion and Formatting Functions

The following GLS API functions provide conversion functions for an internal representation of date and time, as well as formatting functions for an external representation of date and time:

- **ifx_gl_convert_date()**

This function converts the date string stored in *datestr* to an internal representation. The internal representation is stored in the *mi_date* structure to which *date* points.

- **ifx_gl_convert_datetime()**

This function converts the datetime string stored in *datetimestr* to an internal representation. The internal representation is stored in the *mi_datetime* structure to which *datetime* points.

- **ifx_gl_format_date()**

This function uses the format specified by *format* to create a string from the *mi_date* structure to which *date* points. The resulting string is stored in the buffer to which *datestr* points.

- **ifx_gl_format_datetime()**

This function uses the format specified by *format* to create a string from the *mi_datetime* structure to which *datetime* points. The resulting string is stored in the buffer to which *datetimestr* points.

Numeric Conversion and Formatting

The following GLS API functions provide a conversion function for an internal representation of a number string and a formatting function for an external representation of a number string:

- **ifx_gl_convert_number()**

This function converts the number string stored in *decstr* to an internal representation. The internal representation is stored in the *mi_decimal* structure to which *money* points in the format specified by *format*.

- **ifx_gl_format_number()**

This function uses the format specified by *format* to create a string from the value represented by the *mi_number* structure to which *number* points. The resulting string is stored in the buffer to which *numstr* points.

Money Conversion and Formatting

The following GLS API functions provide a conversion function for an internal representation of a money string and a formatting function for an external representation of a money string:

- **ifx_gl_convert_money()**

This function converts the money string stored in *monstr* to an internal representation. The internal representation is stored in the *mi_money* structure to which *money* points.

- **ifx_gl_format_money()**

This function uses the format specified by *format* to create a string from the *mi_money* structure to which *money* points. The resulting string is stored in the buffer to which *monstr* points.

Stream Input and Output

Character data that contains Asian characters must be correctly processed in all graphical user interface (GUI) I/O, clipboard I/O, character terminal I/O, file I/O and network I/O. The following GLS API functions process input and output multibyte character streams:

- **ifx_gl_getmb()**

This function calls the user-defined function *funcp* to obtain bytes that are used to form one multibyte character. This multibyte character is then written to *mb*. The pointer *v* is passed to *funcp* each time that it is called.

- **ifx_gl_putmb()**

This function calls the user-defined function *funcp* with each byte of the multibyte character, *mb*. The pointer *v* is passed to *funcp* each time that it is called.

Initialization and Error Handling

The GLS API provides the following functions for initialization and error handling:

- **ifx_gl_init()**

ESQL/C programs need to call this function at the beginning of **main()** to ensure that the locale, based on the environment variables, has been established before calling any other **ifx_gl_*** functions. This requirement also applies to other client-side programs that use the **ifx_gl_*** functions. However, it is not necessary to call **ifx_gl_init()** from functions that run on the server.

- **ifx_gl_lc_errno()**

GLS library functions use this value, of type *int*, to provide more information about an error that has occurred. This value is set only if an error has occurred, unless it is documented otherwise.

Because any GLS library function can set **ifx_gl_lc_errno()**, you must inspect the error immediately after calling the function in which the error has occurred.

Accessing Messages

Each string that is presented to a user (for example, an error message, informational message, menu item, or button label) should not appear as a literal in a program, but rather as a reference to a message file. In addition, each literal string a user might enter (for example, yes/no responses) should be a reference to a message file.

A literal string does not include program keywords such as `SELECT`, `WHERE`, `IF`, and `WHILE`.

Improving the Performance of Your Programs

The GLS API was created to help the performance of your programs by evaluating the requested GLS API function, determining if the requested function is appropriate for the data that you are trying to process, and then executing the appropriate code.

For example, if you use the `ifx_gl_mbsnext()` function to traverse data that is encoded in a single-byte code set, the function is reduced to a macro that advances the character byte by byte instead of executing code that parses multibyte sequences. Additionally, when collation is based on code-set order rather than locale-defined order, a binary compare (such as `strcmp()`) is used instead of executing algorithms that examine collation weights.

You can also choose how to structure your data to improve performance. Wide-character processing functions are typically faster than multibyte functions. However, wide characters take more space, and as a result, data is generally stored as multibyte strings. If you choose to use wide-character processing, you will have to convert the multibyte strings into wide-character strings, process the strings, and then convert them back to multibyte strings.

This technique is cost-effective if the data you are processing is traversed more than once. The following section describes the GLS API wide-character functions.

Wide-Character String Traversal

A program can traverse a wide-character string in the forward and backward direction. An example is provided for both methods.

Traversing Wide-Character Strings Forward

```
gl_wchar_t buf[], *p;
for ( p = buf; *p != '\0' ; p++ ) process_wchar(*p);
```

Traversing Wide-Character Strings Backward

```
gl_wchar_t buf[], *p;
for ( p = buf + ifx_gl_wcslen(buf) - 1; p >= buf; p-- )
    process_wchar(*p);
```

You can compare or assign a single-byte ASCII character or character constant to a single wide character, as the following example shows:

```
gl_wchar_t wc = 'a';
if ( wc=='a' )
```

Wide-Character String Processing

The following GLS API functions provide wide-character string processing capabilities:

- **ifx_gl_wcscat()**
This function appends a copy of *wcs2* to the end of *wcs1*. If *wcs1* and *wcs2* overlap, the result of this function is undefined.
- **ifx_gl_wcschr()**
This function locates the first occurrence of *wc* in the wide character string *wcs*.
- **ifx_gl_wcscpy()**
This function copies the wide-character string *wcs2* to the location pointed to by *wcs1*. If *wcs1* and *wcs2* overlap, the result of this function is undefined.
- **ifx_gl_wcscspn()**
This function returns the number of characters in the maximum initial substring of *wcs1*, which consists entirely of wide-characters not in the string *wcs2*.

- **ifx_gl_wcslen()**

This function computes the number of wide-character codes in the wide-character string to which *wcs* points, not including the null-terminating wide-character code.

- **ifx_gl_wcsncat()**

This function appends *wcs2* to the end of *wcs1*. No more than *char_limit* characters are read from *wcs2* and written to *wcs1*. If *wcs1* and *wcs2* overlap, the result of this function is undefined.

- **ifx_gl_wcsncpy()**

This function copies *wcs2* to the location pointed to by *wcs1*. No more than *char_limit* characters are read from *wcs2* and written to *wcs1*. If *wcs1* and *wcs2* overlap, the result of this function is undefined.

- **ifx_gl_wcsntslen()**

This function returns the number of characters in *wcs*, not including the trailing space characters. The characters that are not included in the count are the space characters defined in the current locale.

- **ifx_gl_wcsprbk()**

This function searches for the first occurrence in the wide-character string *wcs1* of any wide character from the string *wcs2*.

- **ifx_gl_wcsrchr()**

This function locates the last occurrence of *wc* in the wide-character string *wcs*.

- **ifx_gl_wcsspn()**

This function computes the number of characters in the maximum initial substring of *wcs1*, which consists entirely of wide characters from *wcs2*.

- **ifx_gl_wcswcs()**

This function searches for the first occurrence of the wide-character string *wcs1* in the wide-character string *wcs2*.

Wide-Character Classification

The following GLS API functions test whether the wide character for the respective character classification follows the rules of the current locale:

- **ifx_gl_iswalnum()**
This function returns true if the character is in either the alpha or digit class.
- **ifx_gl_iswalpha()**
This function returns true if the character is an alphabetic character. All uppercase and lowercase characters are also in this class.
- **ifx_gl_iswblank()**
This function returns true if the character is a horizontal space character. The single-byte space and tab characters plus any multibyte version of these characters are in this class.
- **ifx_gl_iswcntrl()**
This function returns true if the character is a control character.
- **ifx_gl_iswdigit()**
This function returns true if the character is a digit. Only the 10 ASCII digits are in this class.
- **ifx_gl_iswgraph()**
This function returns true if the character is a graphical character. All characters that have a visual representation are in this class.
- **ifx_gl_iswlower()**
This function returns true if the character is a lowercase alphabetic character.
- **ifx_gl_iswprint()**
This function returns true if the character is a printable character.
- **ifx_gl_iswpunct()**
This function returns true if the character is a punctuation character. A single-byte ASCII punctuation characters plus any non-ASCII punctuation characters are in this class.

- **ifx_gl_iswspace()**

This function returns true if the character is a horizontal or vertical space character. This class includes all characters from the blank class, single-byte and multibyte version of newline, vertical tab, form feed, and carriage return.

- **ifx_gl_iswupper()**

This function returns true if the character is an upper-case alphabetic character. The ASCII characters *A* through *Z* and all other single and multibyte uppercase characters for Latin-based languages are in this class.

- **ifx_gl_iswxdigit()**

This function returns true if the character is a digit character. Only the 10 ASCII digit characters, *A* through *F*, and *a* through *f* are in this class. Multibyte version or alternative representations (for example, Hindi or Kanji digits) are in the alpha class.

Wide-Character Case Conversion

All alphabetic case conversions must use the conversions specified in the locale. The following functions return the alphabetic case equivalent of the source character, or return the source character if it does not have a case equivalent:

- **ifx_gl_towlower()**

This function converts wide characters to lowercase.

- **ifx_gl_towupper()**

This function converts wide characters to uppercase.

Wide-Character/String Comparison and Sorting

All sorting and comparison of wide characters and wide-character strings must adhere to the comparison order specified in the locale. The **ifx_gl_wcscoll()** function compares wide-character strings *wcs1* and *wcs2* according to the rules of the current locale.

INFORMIX-OnLine Dynamic Server Features

Using the Server Locale	5-3
Generating Non-ASCII Filenames	5-4
The GLS8BITFSYS Environment Variable	5-5
The Server Code Set	5-6
Performing Code-Set Conversion	5-6
Support for Locales by OnLine Utilities.	5-8
Locale Environment Variables.	5-8
Non-ASCII Characters in Command-Line Arguments	5-9
Enhancements to the onmode Utility	5-10

This chapter explains how the Global Language Support (GLS) feature affects INFORMIX-OnLine Dynamic Server. It contains the following topics:

- When OnLine uses the server locale
- How OnLine utilities provide support for the GLS feature

For descriptions of OnLine features that are not unique to the GLS feature, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*, Version 7.2x. For an introduction to the GLS feature, see [Chapter 1, "GLS Fundamentals."](#)

Using the Server Locale

The OnLine database server might perform read and write operations to the following operating-system files:

- Diagnostic files
OnLine generates diagnostic files when you set one or more of the following configuration parameters: DUMPCNT, DUMPCORE, DUMPDIR, DUMPGCORE, and DUMPSHMEM. Diagnostic files include the **af.***, **shmem.xxx**, **gcore.xxx**, and **core** files. For more information on these files, refer to the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.
- Message-log file
OnLine generates a user-specified message-log file when you set the MSGPATH configuration parameter.

Tip: For more information on diagnostic files, message logs, and configuration parameters, see the *"INFORMIX-OnLine Dynamic Server Administrator's Guide."*



These operating-system files reside on the server computer, the same computer on which OnLine resides. When OnLine reads from or writes to these files, it must use a code set that the server computer supports. OnLine obtains this code set from the server locale. You set the server locale with the **SERVER_LOCALE** environment variable. If you do not set **SERVER_LOCALE**, OnLine uses the default locale, U.S. English, as the server locale. For the definition of server locale, see [page 1-27](#). For information on how to set **SERVER_LOCALE**, see [page 2-41](#).

OnLine needs to determine the code set that the server locale supports (the server code set) to perform the following tasks that are associated with the read/write operations to its operating-system files:

- Generate filenames that contain non-ASCII characters
- Perform code-set conversion between the code sets of the server-processing locale and the server locale

Generating Non-ASCII Filenames

You can use non-ASCII characters (8-bit and multibyte characters) when you create or refer to any of the following OnLine names:

- Chunk name
- Message-log filename
- A pathname

The following restrictions affect the ability of OnLine to generate filenames that contain non-ASCII characters:

- OnLine must know whether the operating system is 8-bit clean.
- Your server code set must support these non-ASCII characters.

The GLS8BITFSYS Environment Variable

The **GLS8BITFSYS** environment variable on the server computer tells OnLine whether the operating system is 8-bit clean. OnLine uses the value of **GLS8BITFSYS** to determine whether it can use non-ASCII characters in the filename of an operating-system file that it generates, as follows:

- When **GLS8BITFSYS** is 1, OnLine can use non-ASCII characters in filenames of its operating-system files.
- When **GLS8BITFSYS** is 0, OnLine does *not* allow non-ASCII characters in filenames of its operating-system files.

For example, the MSGPATH configuration parameter specifies a filename for an OnLine message log. Suppose you specify an OnLine message log that is called $/A^1A^2B^1B^2/C^1C^2D^1D^2$, where A^1A^2 , B^1B^2 , C^1C^2 , and D^1D^2 are multibyte characters, with the configuration parameter in your ONCONFIG configuration file that Figure 5-1 shows.

```
MSGPATH /A1A2B1B2/C1C2D1D2  # multibyte message-log filename
```

Figure 5-1
*Message-Log
Filename with
Multibyte
Characters*

If **GLS8BITFSYS** is 1 (or is not set) on the server computer, OnLine assumes that the operating system is 8-bit clean, and it generates a file that is called $C^1C^2D^1D^2$ in the $/A^1A^2B^1B^2$ directory. For more information on an 8-bit clean operating system and the **GLS8BITFSYS** environment variable, see [page 2-23](#).

The Server Code Set

When OnLine actually creates a file whose filename has non-ASCII characters, the server locale must support these non-ASCII characters. Before you start OnLine, you must set the **SERVER_LOCALE** environment variable to the name of a locale whose code set contains these non-ASCII characters.

For example, suppose you want an OnLine message log with the path $/A^1A^2B^1B^2/C^1C^2D^1D^2$, where A^1A^2 , B^1B^2 , C^1C^2 , and D^1D^2 are multibyte characters in the Japanese SJIS code set. You must perform the following steps to enable OnLine to create this message-log file on the server computer:

1. Modify the MSGPATH configuration parameter in the ONCONFIG file, as the example in [Figure 5-1 on page 5-5](#) shows.
2. Set the **SERVER_LOCALE** environment variable on the server computer to the Japanese SJIS locale, **ja_jp.sjis**.
3. Start up OnLine with the **oninit** utility.

When OnLine initializes, it creates the $/A^1A^2B^1B^2/C^1C^2D^1D^2$ message log. For more information on **oninit**, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

Performing Code-Set Conversion

Once OnLine creates the operating-system file, it has generated a filename and written file contents in the code set of the server locale (the server code set). Any Informix product or client application that needs to access this file must have a server-processing locale that supports this same server code set. You must ensure that the appropriate locale environment variables are set so that the server-processing locale supports a code set with these non-ASCII characters. (For more information on the server-processing locale, see [page 1-34](#).)

OnLine checks the validity of a filename with respect to the server-processing locale before it references a filename. When OnLine transfers data to and from its operating-system files, it handles any differences in the code sets of the server-processing locale and the server locale, as follows:

- If these two code sets are the same, OnLine can read from or write to its operating-system files in the code set of the server locale.
- If these two code sets are different, and an Informix code-set conversion exists between them, OnLine automatically performs code-set conversion when it reads from or writes to its operating-system files.

For code-set conversion to resolve the difference in code sets, the server locale must support the actual code set that OnLine used to create the file. For more information, see [“The Server Code Set” on page 5-6](#).

- If these two code sets are different, but no Informix code-set conversion exists, OnLine cannot perform code-set conversion.

If OnLine reads from or writes to an operating-system file for which no code-set conversion exists, it uses the code set of the server-processing locale to perform the read or write operation.

For more information on code-set conversion, see [page 1-40](#).

Support for Locales by OnLine Utilities

The GLS feature enables OnLine utilities to provide the following locale-specific support:

- Locale environment variables to establish nondefault locales
- Non-ASCII characters in command-line arguments

This section provides information that is specific to the use of the GLS feature by OnLine utilities. For a complete description of OnLine utilities, refer to the *INFORMIX-OnLine Dynamic Server Administrator's Guide*. For a complete description of SE utilities, refer to the [Informix Guide to SQL: Reference](#).

Locale Environment Variables

OnLine and SQL utilities are client applications that request information from an instance of OnLine. Therefore, these utilities use the **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables to obtain the name of a nondefault locale, as follows:

- If a database utility is to use a nondefault code set to accept input, (including command-line arguments) and to generate output, you must set the **CLIENT_LOCALE** environment variable.
- If a database utility accesses a database with a nondefault locale, you must set the **DB_LOCALE** environment variable.
- If a database utility causes OnLine to write data on the server computer that has a nondefault code set, you must set the **SERVER_LOCALE** environment variable.

These utilities also perform code-set conversion if the database and the client locales support convertible code sets. For more information on code-set conversion, see [page 1-40](#).

Non-ASCII Characters in Command-Line Arguments

Most OnLine utilities support non-ASCII characters in command-line arguments. These utilities interpret all command-line arguments in the client code set (which `CLIENT_LOCALE` defines). Figure 5-2 shows OnLine utilities that accept non-ASCII characters in command-line arguments.

Figure 5-2
OnLine Utilities That Accept Non-ASCII Characters in Command-Line Arguments

Utility Name	Non-ASCII Characters in Command-Line Argument	Non-ASCII Output
onaudit	<i>-f input_file</i>	Yes
oncheck	<i>-cc -pc database</i> <i>-ci -cI -pk -pK -pl -pL database:table#index_name</i> <i>-ci -cI -pk -pK -pl -pL -cd -cD -pB -pt -pT -pd -pD -pp database:table</i>	Yes
onload	<i>database:table</i> <i>-i old_index new_index</i> <i>-t tape_device</i>	Yes
onlog	<i>-d tape_device</i>	None
onshowaudit	<i>-f input_file</i> <i>-s server_name</i>	Yes
onspaces	<i>-p pathname</i> <i>-f filename</i>	None
onstat	<i>-o filename -dest filename_source</i>	Yes
onunload	<i>database:table</i> <i>-t tape_device</i>	Yes

The DB-Access utility generates labels and messages in the code set of the client locale. The following SQL utilities also accept non-ASCII characters in command-line arguments and generate any output in the client code set:

- **chkenv**
- **dbexport**
- **dbimport**
- **dbload**
- **dbschema**

For a description of these SQL utilities, refer to the [Informix Guide to SQL: Reference](#). For information about how to use these utilities to migrate to or from a database, see the [Informix Migration Guide](#).

Enhancements to the onmode Utility

You can use the **-b** option with the **onmode** utility to revert from a GLS database to an Informix ALS, NLS, or pre-NLS database.

Important: *You cannot revert to an ALS Version 5.0 database from a GLS database.*

For information on how to use the **onmode** utility to migrate to older versions of OnLine, see the [Informix Migration Guide](#). For a complete description of the **onmode** utility, refer to the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.



INFORMIX-SE Features

Using the Server Locale	6-3
Generating Non-ASCII Filenames	6-4
The GLS8BITFSYS Environment Variable	6-4
The Server Code Set	6-5
Data Portability	6-6
Performing Code-Set Conversion	6-6
Naming a Database.	6-7
Support for Locales by SE Utilities	6-8
Locale Environment Variables.	6-8
Non-ASCII Characters in Command-Line Arguments	6-9

This chapter explains how the Global Language Support (GLS) feature affects the INFORMIX-SE database server. It contains the following topics:

- When SE uses the server locale
- How multibyte characters affect the name of an SE database
- How SE utilities provide support for the GLS feature

For descriptions of SE features that are not unique to the GLS feature, see the *INFORMIX-SE Administrator's Guide*. For an introduction to the GLS feature, see [Chapter 1, "GLS Fundamentals."](#)

Using the Server Locale

The SE database server creates operating-system files for the following objects:

- Database
When SE creates a database, it creates a subdirectory that is called **dbname.dbs** in your current directory.
- Table and index (.dat and .idx files)
When you create a table in a database, SE creates two files, a data file (.dat) and an index file (.idx). SE stores the data and index files that are associated with a database in the **dbname.dbs** subdirectory.
- Log file
SE creates this file when you specify the WITH LOG IN clause of the CREATE DATABASE statement.



***Tip:** For more information on database, table, and index files, see the "INFORMIX-SE Administrator's Guide." For more information on the **dbexport** utility, see the ["Informix Migration Guide."](#)*

These operating-system files reside on the server computer, the same computer on which SE resides. When SE reads from or writes to these files, it must use a code set that the server computer supports. SE obtains this code set from the server locale. You set the server locale with the **SERVER_LOCALE** environment variable. If you do not set **SERVER_LOCALE**, SE uses the default locale, U.S. English, as the server locale. For the definition of server locale, see [page 1-27](#). For information on how to set **SERVER_LOCALE**, see [page 2-41](#).

SE needs to determine the code set that the server locale supports (the server code set) to perform the following tasks that are associated with the read/write operations to its operating-system files:

- Generate filenames that contain non-ASCII characters
- Perform code-set conversion between the code sets of the server-processing locale and the server locale

Generating Non-ASCII Filenames

When you specify the name of a database, table, or index, SE creates a file that includes the identifier name. When you create a database log file, SE creates a file with the specified name. The following restrictions affect the ability of SE to generate filenames that contain non-ASCII characters:

- SE must know whether the operating system is 8-bit clean.
- Your server code set must support these non-ASCII characters.

The GLS8BITFSYS Environment Variable

The **GLS8BITFSYS** environment variable on the server computer tells the SE database server whether the operating system is 8-bit clean. SE uses the value of **GLS8BITFSYS** to determine whether it can use non-ASCII characters in the filename of an operating-system file that it generates, as follows:

- When **GLS8BITFSYS** is 1, SE can use non-ASCII characters in its filenames.
- When **GLS8BITFSYS** is 0, SE does *not* allow non-ASCII characters in its filenames.



For example, suppose you create an SE database that is called $A^1A^2B^1B^2$, where A^1A^2 and B^1B^2 are multibyte characters, with the following SQL statement:

```
CREATE DATABASE A1A2B1B2
```

Tip: For more information about how to use multibyte characters in the name of an SE database, see [“Naming a Database” on page 6-7](#).

If **GLS8BITFSYS** is 1 (or is not set) on the server computer, SE assumes that the operating system is 8-bit clean, and it generates a database directory, $A^1A^2B^1B^2$.**.dbs**. For more information on an 8-bit clean operating system and the **GLS8BITFSYS** environment variable, see [page 2-23](#).

The Server Code Set

When SE actually creates a file whose filename has non-ASCII characters, the server locale must support these non-ASCII characters. Before you start SE, you must set the **SERVER_LOCALE** environment variable to the name of a locale whose code set contains these non-ASCII characters.

For example, suppose you want an SE message log with the path $C^1C^2D^1D^2D^3$, where C^1C^2 and $D^1D^2D^3$ are multibyte characters in the Japanese SJIS code set. You must perform the following steps to enable SE to create this message-log file from within a client application:

1. Set the **SERVER_LOCALE** environment variable on the server computer to the Japanese SJIS locale, **ja_jp.sjis**.
2. Set the **CLIENT_LOCALE** environment variable on the client computer to the Japanese SJIS locale, **ja_jp.sjis**, or to a locale that supports a convertible code set. For more information, see [“Performing Code-Set Conversion” on page 6-6](#).
3. Start up the client application from which you execute the following SQL statement:

```
CREATE DATABASE A1A2B1B2 WITH LOG IN 'C1C2D1D2D3'
```

When SE executes this CREATE DATABASE statement, it can support the Japanese SJIS characters in the database name and the message-log file. For more information on CREATE DATABASE, see the [Informix Guide to SQL: Syntax](#).

Data Portability

Use of non-ASCII characters for SE database names and table names might present compatibility problems. To achieve maximum portability for SE, Informix recommends that you not use non-ASCII database names, so that you retain the option to shift the database to an operating system that is not 8-bit clean. To disable the capability of SE to generate 8-bit filenames (even if the operating system is 8-bit clean), set the **GLS8BITFSYS** environment variable to 0.

Performing Code-Set Conversion

Once SE creates the operating-system file, it has generated a filename and written file contents in the code set of the server locale (the server code set). Any Informix product or client application that needs to access this file must have a server-processing locale that supports this same server code set. You must ensure that the appropriate locale environment variables are set so that the server-processing locale supports a code set with these non-ASCII characters. (For more information on the server-processing locale, see [page 1-34](#).)

SE checks the validity of a filename with respect to the server-processing locale before it references a filename. When SE transfers data to and from its operating-system files, it handles any differences in the code sets of the server-processing locale and the server locale as follows:

- If these two code sets are the same, SE can read from or write to its operating-system files in the code set of the server locale.
- If these two code sets are different, and an Informix code-set conversion exists between them, SE automatically performs code-set conversion when it reads from or writes to its operating-system files.

For code-set conversion to resolve the difference in code sets, the server locale must support the actual code set that SE used to create the file. For more information, see [“The Server Code Set” on page 6-5](#).

- If these two code sets are different, but no Informix code-set conversion exists, SE cannot perform code-set conversion.

If SE reads from or writes to an operating-system file for which no code-set conversion exists, it uses the code set of the server-processing locale to perform the read or write operation.

For more information on code-set conversion, see [page 1-40](#).

Naming a Database

The SE database server limits the name of a database to 10 bytes. When you use multibyte characters in database names, you must ensure that the database name meets this size requirement. The following CREATE DATABASE statement creates a database name of five multibyte characters:

```
CREATE DATABASE A1A2B1B2C1C2D1D2E1E2
```

The database name that is shown in the preceding representation is 10 bytes long (five 2-byte multibyte characters), so it does not exceed the maximum length for SE database names. However, the following CREATE DATABASE statement generates an error because the total number of bytes in this database name is 12:

```
CREATE DATABASE A1A2A3A4B1B2B3B4C1C2C3C4
```

This statement specifies three 4-byte characters for the name of an SE database. In this case, the total number of bytes in the database name would be 12 bytes, and this length exceeds the maximum length for a database name in SE. The database server rejects the statement with the following error message:

```
-354: Incorrect database or cursor name format
```

To recover from this error, you must reenter the CREATE DATABASE statement and specify a name whose total length in the buffer is not more than 10 bytes.



Tip: You specify the database name in the code set of the client locale (the client code set). When your client application performs code-set conversion on a database name, it might expand this name beyond the 10-character limit. Make sure that the database name is 10 or fewer characters that the database code set defines, not just that the client code set defines.

Support for Locales by SE Utilities

The GLS feature enables SE utilities to provide the following locale-specific support:

- Locale environment variables to establish nondefault locales
- Non-ASCII characters in command-line arguments

This section provides information specific to the use of the GLS feature by SE utilities. For a complete description of SE utilities, refer to the [Informix Guide to SQL: Reference](#). For information about how to use these utilities to migrate to or from a database, see Chapter 8 of the [Informix Migration Guide](#).

Locale Environment Variables

SE and SQL utilities are client applications that request information from an SE database server. Therefore, these utilities use the **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables to obtain the name of a nondefault locale, as follows:

- If a database utility is to use a nondefault code set to accept input, (including command-line arguments) and to generate output, you must set the **CLIENT_LOCALE** environment variable.
- If a database utility accesses a database with a nondefault locale, you must set the **DB_LOCALE** environment variable.
- If a database utility causes SE to write data on the server computer that has a nondefault code set, you must set the **SERVER_LOCALE** environment variable.

These utilities also perform code-set conversion if the database and the client locales support convertible code sets. For more information on code-set conversion, see [page 1-40](#).

Non-ASCII Characters in Command-Line Arguments

SE utilities support non-ASCII characters in command-line arguments. These utilities interpret all command-line arguments in the client code set (that **CLIENT_LOCALE** defines). Figure 6-1 shows SE utilities that accept non-ASCII characters in command-line arguments.

Figure 6-1
SE Utilities That Accept Non-ASCII Characters in Command-Line Arguments

Utility Name	Non-ASCII Characters in Command-Line Arguments	Non-ASCII Output
secheck	<i>filename</i>	None
selog	<i>log filename</i> <i>-f tablename</i>	None

The SE utilities accept non-ASCII characters in the parameters for *filename*, *log filename*, and *tablename*.

The DB-Access utility generates labels and messages in the code set of the client locale. The following SQL utilities also accept non-ASCII characters in command-line arguments and generate any output in the client code set:

- **dbexport**
- **dbimport**
- **dbload**
- **dbschema**

For a description of these SQL utilities, refer to the [Informix Guide to SQL: Reference](#). For information about how to use these utilities to migrate to or from a database, see the [Informix Migration Guide](#).

General SQL API Features

Non-ASCII Characters in ESQL Source Files	7-3
Generating Non-ASCII Filenames	7-4
The GLS8BITFSYS Environment Variable	7-5
The Client Code Set	7-5
Using Non-ASCII Characters in Source Code	7-6
Enhanced ESQL Library Functions	7-7
DATE-Format Functions	7-8
GL_DATE Support	7-8
DBDATE Extensions.	7-9
Extended DATE-Format Strings.	7-10
Precedence for Date End-User Formats	7-14
DATETIME-Format Functions	7-15
GL_DATETIME Support	7-15
DBTIME Support.	7-16
Extended DATETIME-Format Strings.	7-16
Precedence for DATETIME End-User Formats.	7-17
Numeric-Format Functions.	7-17
Support for Multibyte Characters	7-18
Locale-Specific Numeric Formatting	7-18
Currency-Symbol Formatting	7-20
DBMONEY Extensions.	7-22
String Functions	7-23
GLS-Specific Error Messages	7-23
Establishing a Database Connection	7-24
Sending Client-Locale Information	7-24
Checking for Connection Warnings.	7-24
Avoiding Partial Characters	7-25
Copying Character Data.	7-26
Using Code-Set Conversion	7-26



This chapter explains how the Global Language Support (GLS) feature affects the Informix SQL application programming interfaces (APIs), INFORMIX-ESQL/C and INFORMIX-ESQL/COBOL. It contains the following topics:

- Where multibyte characters are valid in an embedded-language (ESQL) source file
- How ESQL library functions have been enhanced to support multibyte characters and localized date and time end-user formats
- How to check for warnings within an ESQL application if a connection encounters locale-related inconsistencies

Tip: This chapter covers GLS information for both ESQL/C and ESQL/COBOL. For information specific to ESQL/C, refer to [Chapter 8, “INFORMIX-ESQL/C Features.”](#)

For descriptions of Informix SQL API features that are not unique to the GLS feature, see the [INFORMIX-ESQL/C Programmer's Manual](#).

Non-ASCII Characters in ESQL Source Files

Each Informix SQL API product contains a processor to process an ESQL source file that has embedded SQL and preprocessor statements:

- The ESQL/C processor, **esql**, processes C source files.
- The ESQL/COBOL processor processes COBOL source files.
The name of the ESQL/COBOL processor is **esqlcobol**. ♦
The name of the ESQL/COBOL processor is **esqlcob**. ♦

UNIX

Windows

The processors for ESQL/C and ESQL/COBOL products use operating-system files in the following situations:

- They write language-specific source files (.c and .cob files) when they process an ESQL source file.

The ESQL processors use the client code set (that the client locale specifies) to generate the filenames for these language-specific files.

- They read ESQL source files (.ec and .eco) that the user creates.

The ESQL processors use the client code set to interpret the contents of these ESQL source files.

You specify the client locale with the **CLIENT_LOCALE** environment variable. For more information on this environment variable, see [page 2-6](#).

Generating Non-ASCII Filenames

When an ESQL processor processes an ESQL source file, it must generate a corresponding intermediate file for the source file. If you use non-ASCII characters (8-bit and multibyte character) in these source-file names, the following restrictions affect the ability of the ESQL/C or ESQL/COBOL processor to generate filenames that contain non-ASCII characters:

- The ESQL processor must know whether the operating system is 8-bit clean.
- The code set of the client locale (the client code set) must support the non-ASCII characters that are used in the ESQL source filename.
- Your C or COBOL compiler supports the non-ASCII characters that the filename of the ESQL/C or ESQL/COBOL source file uses.

If your C compiler does not support non-ASCII characters, you can use the **CC8BITLEVEL** environment variable as a workaround when your source file contains multibyte characters. For more information, see [“The esqlmf Filter” on page 8-7](#).

The GLS8BITFSYS Environment Variable

The **GLS8BITFSYS** environment variable on the client computer tells an ESQL processor whether the operating system is 8-bit clean. An ESQL processor uses the value of **GLS8BITFSYS** to determine whether it can use non-ASCII characters in the filename of an ESQL intermediate file that it generates, as follows:

- When **GLS8BITFSYS** is 1, an ESQL processor can use non-ASCII characters in filenames that it generates.
- When **GLS8BITFSYS** is 0, an ESQL processor does *not* allow non-ASCII characters in filenames that it generates.

For example, suppose you compile an ESQL/C source file that is called $A^1A^2B^1B^2$.ec, where A^1A^2 and B^1B^2 are multibyte characters. If **GLS8BITFSYS** is set to 1 (or is not set) on the client computer, an ESQL processor assumes that the operating system on the client computer is 8-bit clean, and it generates an intermediate C file that is called $A^1A^2B^1B^2$.c.

For more information on an 8-bit-clean operating system and on the **GLS8BITFSYS** environment variable, see [page 2-23](#).

The Client Code Set

When an ESQL processor actually creates a file whose filename has non-ASCII characters, the client locale must support these non-ASCII characters. Before you start an Informix database server, you must ensure that the code set of the client locale (the client code set) contains these characters. When you use a nondefault locale, you must set the **CLIENT_LOCALE** environment variable to the name of a locale whose code set contains these non-ASCII characters.

For example, suppose you want to process an ESQL/C source file with the path $/A^1A^2B^1B^2/C^1C^2D^1D^2$, where A^1A^2 , B^1B^2 , C^1C^2 , and D^1D^2 are multibyte characters in the Japanese SJIS code set. You must perform the following steps to enable the **esql** command to create the intermediate C source file on the client computer:

1. Set the **CLIENT_LOCALE** environment variable on the client computer to the Japanese SJIS locale, **ja_jp.sjis**.
2. Process the ESQL/C source file with the **esql** command.

Once an ESQL processor creates an intermediate file, it has generated a filename and written file contents in the client code set. Any Informix product or client application that needs to access this file must have a client locale that supports this same code set.

If the code sets that are associated with the filename and with the client locale do not match, a valid filename might contain illegal characters with respect to the client locale. The ESQL processor rejects any filename that contains illegal characters and displays the following error message:

```
Illegal characters in filename.
```

Using Non-ASCII Characters in Source Code

The ESQL processors obtain the code set for use in ESQL source files from the client locale. Within an ESQL source file, you can use non-ASCII characters for the following objects (these examples use A^1A^2 and B^1B^2 to represent multibyte characters):

- ESQL host variable and indicator variable names

For example, in an ESQL/C program, the following use of multibyte characters is valid:

```
char A1A2[20], B1B2[20];  
:  
:  
EXEC SQL select col1, col2 into :A1A2 :B1B2;
```

For more information on ESQL/C host variables, see [“Non-ASCII Characters in Host Variables” on page 8-4](#).

- ESQL comments

For example, in an ESQL/COBOL program, the following use of multibyte characters is valid:

```
EXEC SQL CONNECT TO A1A2 END-EXEC. -- A1A2 DATABASE OPENED
```

- Names of SQL identifiers such as databases, tables, columns, views, constraints, prepared statements and cursors

For more information, see [“Naming Database Objects” on page 3-4](#).



- **Literal strings**

For example, in an ESQL/C program, the following use of multibyte characters is valid:

```
EXEC SQL insert into tbl1 (nchr1) values 'A1A2B1B2';
```

- **Filenames and pathnames, as long as your operating system supports multibyte characters in filenames and pathnames**

For example, in an ESQL/COBOL program, the following use of multibyte characters is valid:

```
EXEC SQL INCLUDE 'A1A2/B1B2' END-EXEC.
```

***Tip:** Some C-language compilers support multibyte characters in literals or comments only. For such compilers, you might need to set the **ESQLMF** and **CC8BITLEVEL** environment variables so that the ESQL/C processor calls a multibyte filter. For more information, see [“The esqlmf Filter” on page 8-7](#).*

To use non-ASCII characters in your ESQL source file, the client locale must support them.

Enhanced ESQL Library Functions

Informix SQL API products support locale-specific enhancements to the ESQL/C library functions and ESQL/COBOL routines. These ESQL library functions fall into the following categories:

- DATE-format functions
- DATETIME-format functions
- Numeric-format functions
- String functions

In addition, this section describes the GLS-related error messages that these ESQL functions might produce.

DATE-Format Functions

The ESQL DATE-format functions in Figure 7-1 support the following extensions to format era-based DATE values:

- Support for the `GL_DATE` environment variable
- Support for era-based date formats of the `DBDATE` environment variable
- Extensions to the date-format strings for ESQL DATE-format functions
- Support for a precedence of date end-user formats

ESQL/C Date Function	ESQL/COBOL Date Routine
<code>rdatestr()</code>	<code>ECO-DAT</code>
<code>rstrdate()</code>	<code>ECO-STR</code>
<code>rdefmtdate()</code>	<code>ECO-DEF</code>
<code>rfmtdate()</code>	<code>ECO-FMT</code>

Figure 7-1
*ESQL DATE-Format Functions That
Support Era-Based Extensions*

This section describes locale-specific behavior of the ESQL DATE-format functions. For general information on the syntax and use of the ESQL/C DATE-format functions, see Chapter 6 of the [INFORMIX-ESQL/C Programmer's Manual](#). For general information on the syntax and use of ESQL/COBOL DATE-format routines, see Chapter 3 of the *INFORMIX-ESQL/COBOL Programmer's Manual*.

GL_DATE Support

The value of the `GL_DATE` environment variables can affect the results that these ESQL DATE-format functions generate. The end-user format that `GL_DATE` specifies overrides date end-user formats that the client locale defines. For more information, see [“Precedence for Date End-User Formats” on page 7-14](#).

DBDATE Extensions

Figure 7-2 shows the ESQL DATE-format functions that support the extended era-based date syntax for the **DBDATE** environment variable. (For more information of the era-based date extensions, see the description of **DBDATE** on [page 2-7](#).)

ESQL/C Date Function	ESQL/COBOL Date Routine
rdatestr()	ECO-DAT
rstrdate()	ECO-STR

Figure 7-2
*ESQL DATE-Format Functions That
Support DBDATE Extensions*

When you set **DBDATE** to one of the era-based formats, these functions use era-based dates to convert between date strings and internal DATE values. The following ESQL/C example shows a call to the **rdatestr()** library function:

```
char str[100];
long jdate;
:
rdatestr(jdate, str);
printf("%s\n", str);
```

If you set **DBDATE** to GY2MD/ and **CLIENT_LOCALE** to the Japanese SJIS locale (**ja_jp.sjis**), the preceding code fragment prints the following value for the date 08/18/1990:

```
H02/08/18
```



Important: Informix products treat any undefined characters in the alphabetic era specification as an error.

If you set **DBDATE** to a era-based date format (that is specific to a Chinese or Japanese locale), make sure to set the **CLIENT_LOCALE** environment variable to a locale that supports era-based dates.

Extended DATE-Format Strings

Figure 7-3 shows the ESQL DATE-format functions that support the extended-DATE format strings.

ESQL/C Date Function	ESQL/COBOL Date Routine
rdefmtdate()	ECO-DEF
rfmtdate()	ECO-FMT

Figure 7-3
*ESQL DATE-Format Functions That
Support Extended-Format Strings*

Figure 7-4 shows the extended-format strings that these ESQL functions support for use with GLS locales.

Figure 7-4 <i>Extended-Format Strings for GLS Locales</i>		
Era Year	Format	Era Used
Full era year: full name of the base year (period) followed by a 2-digit year offset Same as GL_DATE end-user format of "%EC%02.2Ey"	eyy	The era that the client locale (that CLIENT_LOCALE indicates) defines
Abbreviated era year: abbreviated name of the base year (period) followed by a 2-digit year offset Same as GL_DATE end-user format of "%Eg%02.2Ey"	gyy	The era that the client locale (that CLIENT_LOCALE indicates) defines

The extended-format strings in Figure 7-4 format eras with 2-digit year offsets. To obtain a 4-digit year offset, use the **c1**, **j1**, or **j2** extended-format strings (see [Figure 7-6 on page 7-12](#)).

Figure 7-5 shows some sample extended-format strings for era-based dates.

Figure 7-5
Sample Extended-Format Strings for GLS Locales

Description	Sample Format	October 5, 1990 prints as:
Abbreviated era year	gyymmdd	H021005
	gyy.mm.dd	H02.10.05
Full era year	eyymmdd	A ¹ A ² 021005
	eyy-mm-dd	A ¹ A ² 02-10-05
	eyyB ¹ B ² mmB ¹ B ² ddB ¹ B ²	A ¹ A ² 02B ¹ B ² 10B ¹ B ² 05B ¹ B ²

The examples in Figure 7-5 assume that the client locale is Japanese SJIS (**ja_jp.sjis**).

The following ESQL/C code fragment contains a call to the **rdefmtdate()** library function:

```
char fmt_str[100];
char in_str[100];
long jdate;
:
:
rdatestr("10/05/95", &jdate);
strcpy("gyy/mm/dd", fmt_str);
rdefmtdate(&jdate, fmt_str, in_str);
printf("Abbreviated Era Year: %s\n", in_str);

strcpy("eyymmdd", fmt_str);
rdefmtdate(&jdate, fmt_str, in_str);
printf("Full Era Year: %s\n", in_str);
```

When the **CLIENT_LOCALE** specifies the Japanese SJIS (**ja_jp.sjis**) locale, the code fragment displays the following output:

```
Abbreviated Era Year: H07/10/05
Full Era Year: H021005
```

ALS

Informix Version 9.1 products support additional extended-format strings for backward compatibility with Informix ALS products. Informix recommends that you use the extended-format strings that work with GLS locales (see [Figure 7-4 on page 7-10](#)) in Version 9.1 client applications.

Figure 7-6 shows the extended-format strings that these ESQL functions support for backward compatibility with ALS Version 5.0 products.

Figure 7-6
Extended-Format Strings for ALS Version 5.x Compatibility

Era Year	Format	Era Used
Full era year: full name of the base year (period) followed by a 2-digit year offset Same as GL_DATE end-user format of "%EC%02.2Ey"	yyj2	Japanese Imperial
Full era year: full name of the base year (period) followed by a 4-digit year offset Same as GL_DATE end-user format of "%EC%04.4Ey"	yyyyj2	Japanese Imperial
Abbreviated era year: abbreviated name of the base year (period) followed by a 2-digit year offset Same as GL_DATE end-user format of "%Eg%02.2Ey"	yyc1 yyj1	Taiwanese Ming Guo Japanese Imperial
Abbreviated era year: abbreviated name of the base year (period) followed by a 4-digit year offset Same as GL_DATE end-user format of "%Eg%04.4Ey"	yyyc1 yyyyj1	Taiwanese Ming Guo Japanese Imperial

The following table shows some sample ALS Version 5.0 extended-format strings for era-based dates.

Description	Sample Format	October 5, 1990 prints as:
Japanese era:		
abbreviated era year, 2-digit year	yymmddj1	H021005
abbreviated era year, 4-digit year	yyyymmddj1	H00021005
full era year, 2-digit year	yymmddj2	A ¹ A ² 021005
full era year, 4-digit year	yyyymmddj2	A ¹ A ² 00021005
Taiwanese Ming Guo date:		
with 2-digit year	yymmddc1	791005
	yy.mm.ddc1	79.10.05
	yyA ¹ A ² mmA ¹ A ² ddA ¹ A ²	79A ¹ A ² 10A ¹ A ² 05A ¹ A ²
with 4-digit year	yyyymmddc1	00791005

Figure 7-7 shows the extended-format strings that these ESQL functions support for backward compatibility with ALS Version 4.x products.

Figure 7-7
Extended-Format Strings for ALS 4.x Compatibility

Era Format	Format	Locale
Full era year: full name of the base year (period) followed by a 2-digit year offset Same as GL_DATE end-user format of "%EC%02.2Ey"	nnnnyy	Japanese
Abbreviated era year: abbreviated name of the base year (period) followed by a 2-digit year offset Same as GL_DATE end-user format of "%Eg%02.2Ey"	nyy	Japanese
Day of the week, with symbols that the locale defines (Sunday through Saturday in Asian characters). Multibyte characters are printed if DB_LOCALE is set to another language.	ww	

The following table shows some sample ALS Version 4.x extended-format strings for era-based dates.

Description	Sample Format	October 5, 1990 prints as:
Japanese era:		
abbreviated era year	nyymmdd	H021005
full era year	nnnnyy mm dd nnnnyymmdd (ww)	A ¹ A ² 02 10 05 A ¹ A ² 021005 (B ¹ B ²)

In addition to the era-based combinations in [Figure 7-4 on page 7-10](#) through [Figure 7-7 on page 7-13](#), the format string also supports the combinations that the *INFORMIX-ESQL/C Programmer's Manual* and the *INFORMIX-ESQL/COBOL Programmer's Manual* describe for these functions.

Precedence for Date End-User Formats

The ESQL DATE-format functions use the following precedence to determine the end-user format for values in DATE columns:

1. The end-user format that **DBDATE** specifies (if **DBDATE** is set)
2. The end-user format that **GL_DATE** specifies (if **GL_DATE** is set)
3. The date end-user format that the client locale specifies (if **CLIENT_LOCALE** is set)
4. The date end-user form from the default locale: **%m %d %iY**

For more information on the precedence of **DBDATE**, **GL_DATE** and **CLIENT_LOCALE**, refer to “[Customizing Date and Time End-User Formats](#)” on page 1-50.



***Tip:** Informix products support **DBDATE** for backward compatibility with earlier products. Informix recommends the use of the **GL_DATE** environment variable for new client applications.*

DATEIME-Format Functions

The ESQL DATEIME-format functions in Figure 7-8 support the following extensions to format era-based DATEIME values:

- Support for the **GL_DATEIME** environment variable
- Support for era-based date and times of the **DBTIME** environment variable
- Extensions to the date and time format strings for ESQL DATEIME-format functions
- Support for a precedence of DATEIME end-user formats

ESQL/C DATEIME Function	ESQL/COBOL DATEIME Routine
dtevfmtasc()	ECO-DTCVASC
dttofmtasc()	ECO-DTTOASC

Figure 7-8
*ESQL DATEIME-Format Functions
That Support Era-Based Extensions*

This section describes locale-specific behavior of the ESQL DATEIME-format functions. For general information on the syntax and use of the ESQL/C DATEIME-format functions, see Chapter 6 of the [INFORMIX-ESQL/C Programmer's Manual](#). For general information on the syntax and use of ESQL/COBOL DATEIME-format routines, see Chapter 3 of the *INFORMIX-ESQL/COBOL Programmer's Manual*.

GL_DATEIME Support

The value of the **GL_DATEIME** environment variables can affect the results that these ESQL DATEIME-format functions generate. The end-user format that **GL_DATEIME** specifies overrides date and time formats that the client locale defines. For more information, see [“Precedence for DATEIME End-User Formats” on page 7-17](#).

DBTIME Support

The ESQL DATETIME-format functions in [Figure 7-8 on page 7-15](#) support the extended era-based date and time format strings for the **DBTIME** environment variable. (For more information on the era-based date and time extensions, see the description of **DBTIME** on [page 2-20](#).)

When you set **DBTIME** to one of the era-based formats, these functions can use era-based dates and times to convert between literal DATETIME strings and internal DATETIME values.

***Tip:** Informix products support **DBTIME** for backward compatibility with earlier products. Informix recommends the use of the **GL_DATETIME** environment variable for new applications.*

If you set **DBTIME** to a era-based DATETIME format (that is specific to a Chinese or Japanese locale), make sure to set the **CLIENT_LOCALE** environment variable to a locale that supports era-based dates and times.

Extended DATETIME-Format Strings

The following table shows the extended-format strings that the ESQL functions in [Figure 7-8 on page 7-15](#) support.

Format	Description	December 27, 1991 appears as:
%y %m %dc1	Taiwanese Ming Guo date	80 12 27
%Y %m %dc1	Taiwanese Ming Guo date	0080 12 27
%y %m %dj1	Japanese era with abbreviated era symbols	H03 12 27
%Y %m %dj1	Japanese era with abbreviated era symbols	H0003 12 27
%y %m %dj2	Japanese era with full era symbols	A ¹ A ² B ¹ B ² 03 12 27
%Y %m %dj2	Japanese era with full era symbols	A ¹ A ² B ¹ B ² 0003 12 27

In addition to the formats in the preceding table, these ESQL DATETIME-format functions support the GLS date and time specifiers. For a list of these specifiers, see the description of the **GL_DATE** and **GL_DATETIME** environment variables in [Chapter 2, “GLS Environment Variables.”](#)



Precedence for DATETIME End-User Formats

The ESQL DATETIME-format functions use the following precedence to determine the end-user format of values in DATETIME columns:

- 1. The end-user format that **DBTIME** specifies (if **DBTIME** is set)
- 2. The end-user format that **GL_DATETIME** specifies (if **GL_DATETIME** is set)
- 3. The date and time end-user formats that the client locale specifies (if **CLIENT_LOCALE** is set)
- 4. The date and time end-user format from the default locale:
 %iY-%m-%d %H:%M:%S

For more information on the precedence of **DBDATE**, **GL_DATE** and **CLIENT_LOCALE**, refer to [“Customizing Date and Time End-User Formats” on page 1-50](#).

Numeric-Format Functions

The ESQL numeric-format functions in Figure 7-9 support the following extensions to format numeric values:

- Support for multibyte characters in format strings
- Locale-specific formats for numeric values
- Formatting characters for currency symbols
- Support for the **DBMONEY** environment variable

ESQL/C Numeric Function	ESQL/COBOL Numeric Routine
rfmtdec()	None
rfmtdouble()	ECO-FEL
rfmtlong()	ECO-FIN

Figure 7-9
ESQL Numeric-Format Functions That Support Extended Numeric Formats



This section describes locale-specific behavior of the ESQL numeric-format functions. For general information on the syntax and use of the ESQL/C numeric-format functions, see Chapter 5 of the [INFORMIX-ESQL/C Programmer's Manual](#). For general information on the syntax and use of ESQL/COBOL numeric-format routines, see Chapter 2 of the *INFORMIX-ESQL/COBOL Programmer's Manual*.

Tip: For a list of errors that these ESQL numeric-format functions might return, see [“GLS-Specific Error Messages” on page 7-23](#).

Support for Multibyte Characters

The ESQL numeric-format functions in [Figure 7-9 on page 7-17](#) support multibyte characters in their format strings as long as your client locale supports a multibyte code set that defines these characters. However, these ESQL functions and routines interpret multibyte characters as literal characters. You cannot use multibyte equivalents of the ASCII formatting characters.

For example, the following ESQL/C code fragment shows a call to the **rfmtlong()** function with the multibyte character A¹A² in the format string:

```
stcopy("A1A2***,***", fmtbuf);
rfmtlong(78941, fmtbuf, outbuf);
printf("Formatted value: %s\n", outbuf);
```

This code fragment generates the following output (as long as the client code set contains the A¹A² character):

```
Formatting value: A1A2*78,941
```

Locale-Specific Numeric Formatting

The ESQL numeric-format functions in [Figure 7-9 on page 7-17](#) require a format string as an argument. This format string determines how the numeric-format function formats the numeric value. A format string consists of a series of formatting characters and the following currency notation.

Formatting Character	Function
Dollar sign (\$)	Currency symbol
Comma (,)	Thousands separator
Period (.)	Decimal separator

Regardless of the client locale that you use, you must use the preceding ASCII symbols in the format string to identify where to place the currency symbol, decimal separator, and thousands separator. The numeric-format function uses the following precedence to translate these symbols to their locale-specific equivalents:

1. The symbols that **DBMONEY** indicates (if **DBMONEY** is set)
For locale-specific behavior of **DBMONEY**, see [“DBMONEY Extensions” on page 7-22](#).
2. The symbols that the appropriate locale category of the client locale (if **CLIENT_LOCALE** is set) specifies
If the format string contains either a \$ or @ formatting character, a numeric-format function assumes that the value is a monetary value and refers to the **MONETARY** category of the client locale. If these two symbols are not in the format string, a numeric-format function refers to the **NUMERIC** category of the client locale.
For more information on the use of the \$ and @ formatting characters, see [“Currency-Symbol Formatting” on page 7-20](#). For more information on the **MONETARY** and **NUMERIC** locale categories, see [“Locale Categories” on page A-4](#).
3. The actual symbol that appears in the format string (\$, comma, or period)

In other words, these numeric-format functions replace the dollar sign in the format string with the currency symbol that **DBMONEY** specifies (if it is set) or with the currency symbol that the client locale specifies (if **DBMONEY** is *not* set). The same is true for the decimal separator and thousands separator.

For example, the following ESQL/C code fragment shows a call to the **rfmtlong()** function:

```
stcopy("$***,***.&&", fmtbuf);
rfmtlong(78941, fmtbuf, outbuf);
printf("Formatted value: %s\n", outbuf);
```

In the default, German, and Spanish locales, this code fragment produces the following results for the internal MONEY value of 78941.00 (if **DBMONEY** is not set).

Format String	Client Locale	Formatted Value
\$***,***.&&	Default locale (en_us.8859-1)	\$*78,941.00
	German locale (de_de.8859-1)	DM*78.941,00
	Spanish locale (es_es.8859-1)	Pts*78.941,00

Currency-Symbol Formatting

The ESQL numeric-format functions in [Figure 7-9 on page 7-17](#) support all formatting characters that Chapter 5 of the *INFORMIX-ESQL/C Programmer's Manual* and Chapter 2 of the *INFORMIX-ESQL/COBOL Programmer's Manual* describe. In addition, you can use the following formatting characters to indicate the placement of a currency symbol in the formatted output.

Formatting Character	Function
\$	This character is replaced by the <i>precede-currency symbol</i> , if the locale defines one. The MONETARY category of the locale defines the precede-currency symbol, which is the symbol that appears <i>before</i> a monetary value. When you group several dollar signs in a row, a single currency symbol floats to the rightmost position that it can occupy without interfering with the number.
@	This character is replaced by the <i>succeed-currency symbol</i> , if the locale defines one. The MONETARY category of the locale defines the succeed-currency symbol, which is the symbol that appears <i>after</i> a monetary value.

For more information, see [“The MONETARY Category” on page A-7](#).

You can include both formatting characters in a format string. The locale defines whether the currency symbol appears before or after the monetary value, as follows:

- If the locale formats monetary values with a currency symbol *before* the value, the locale sets the currency symbol to the precede-currency symbol and sets the succeed-currency symbol to a blank character.
- If the locale formats monetary values with a currency symbol *after* the value, the locale sets the currency symbol to the succeed-currency symbol and sets the precede-currency symbol to a blank character.

The default locale defines the currency symbol as the precede-currency symbol, which appears as a dollar sign (\$). In the default locale, the succeed-currency symbol appears as a blank. In the default, German, and French locales, the numeric-format functions produce the following results for the internal MONEY value of 1.00.

Format String	Client Locale	Formatted Result
\$***,***	Default locale (en_us.8859-1)	\$*****1
	German locale (de_de.8859-1)	DM*****1
	French locale (fr_fr.8859-1)	s*****1
\$***,***@	Default locale (en_us.8859-1)	\$*****1s
	German locale (de_de.8859-1)	DM*****1s
	French locale (fr_fr.8859-1)	s*****1FF
\$\$,\$\$\$.\$\$	Default locale (en_us.8859-1)	ssss\$1.00
	German locale (de_de.8859-1)	ssssDM1,00
	French locale (fr_fr.8859-1)	sssss1FF
,@	Default locale (en_us.8859-1)	*****1s
	German locale (de_de.8859-1)	*****1s
	French locale (fr_fr.8859-1)	*****1FF
@***,***	Default locale (en_us.8859-1)	s*****1
	German locale (de_de.8859-1)	s*****1
	French locale (fr_fr.8859-1)	FF*****1

In the preceding table, the character *s* represents a blank or space, the FF is the French currency symbol for French francs, and the DM is the German currency symbol for deutsche marks. For more information on locale-specific currency notation, see [page 7-18](#).

The **DBMONEY** environment variable can also set the precede-currency symbol and the succeed-currency symbol. The syntax diagram for **DBMONEY** (see [page 2-19](#)) refers to these symbols as *front* and *back*, respectively. If set, **DBMONEY** takes precedence over the symbols that the locale defines.

DBMONEY Extensions

You can specify the currency symbol and decimal-separator symbol with the **DBMONEY** environment variable. These settings override any currency notation that the client locale specifies.

You can use multibyte characters for these symbols, as long as your client code set supports them. For example, the following table shows how multibyte characters appear in sample output.

Format String	Number to be Formatted	DBMONEY	Output
"\$\$,\$\$\$.\$\$"	1234	'\$'.	\$1,234.00
"\$\$,\$\$\$.\$\$"	1234	DM,	DM1.234,00
"\$\$,\$\$\$.\$\$"	1234	A ¹ A ² .	A ¹ A ² 1,234.00
"\$\$,\$\$\$.\$\$"	1234	.A ¹ A ²	s1,234.00
"&&,&&&.&&@"	1234	.A ¹ A ²	s1,234.00A ¹ A ²
"\$&&,&&&.&&@"	1234	A ¹ A ² .	A ¹ A ² s1,234.00
"\$&&,&&&.&&@"	1234	.A ¹ A ²	s1,234.00A ¹ A ²
"@&&,&&&.&&@"	1234	.A ¹ A ²	A ¹ A ² s1,234.00

In the preceding table, the character *s* represents a blank or space. For more information on **DBMONEY**, see [page 2-19](#).

String Functions

Figure 7-10 shows the ESQL string functions that support locale-specific shifted characters.

ESQL/C String Function	ESQL/COBOL String Routine
rdownshift()	ECO-DSH
rupshift()	ECO-USH

Figure 7-10
*ESQL String Functions That Support
Locale-Specific Shifted Characters*

These string functions use the information in the CTYPE category of the client locale to determine the shifted code points. (For more information on the CTYPE locale category, see [page A-6](#).) If the client locale specifies a multibyte code set, these functions can operate on multibyte strings.

Important: *With multibyte character strings, a shifted string might occupy more memory after a shift operation than it did before. You must ensure that the buffer you pass to these ESQL shift functions is large enough to accommodate this expansion.*



GLS-Specific Error Messages

The following ESQL functions might generate GLS-specific error messages:

- DATE-format functions (see [Figure 7-1 on page 7-8](#))
- DATETIME-format functions (see [Figure 7-8 on page 7-15](#))
- Numeric-format functions (see [Figure 7-9 on page 7-17](#))

For more information on GLS-specific error messages, refer to the [Informix Error Messages](#) manual.

Establishing a Database Connection

To connect to a database, an ESQL client application requests a connection from the database server. The database server must verify that it can access the database and establish the connection between the client and the database. Your client application performs the following tasks:

- Sends its client and database locale information to the database server
The ESQL program performs this step automatically when it requests a connection.
- Checks for connection warnings that the database server generates
You must provide code in your ESQL program to perform this step.

Sending Client-Locale Information

When an ESQL client program requests a connection to a database, it sends its client-locale information to the database server. The database server uses this information to determine the following information:

- How are numeric and monetary values formatted?
- How are dates and times formatted?
- What database locale does the client expect?

For information on the subset of client-locale information that the client program sends, see [“Sending the Client Locale” on page 1-32](#).

Checking for Connection Warnings

The ESQL client application sends the value of its database locale (DB_LOCALE) to the database server. The database server must verify that this database is compatible with the database locale in the system catalog of the database that the client application wants to open.

If this comparison fails, the database server sets the SQLWARN7 warning flag in the SQLCA structure and uses the database locale in the database system catalog. An ESQL client application can check for this warning as follows:

- An ESQL/C client application checks the **sqlca.sqlwarn.sqlwarn7** field.
- An ESQL/COBOL client application checks the SQLWARN7 OF SQLWARN OF SQLCA field.

If the **sqlwarn7** field has a value of W, the database server has ignored the database locale that the client specified and has instead used the locale in the database as the database locale.



Warning: *If the database server generates a warning when it establishes the connection, your application should check the SQLCA structure for warnings after a connection. If you proceed with such a connection, it is your responsibility to understand the format of the data that is being exchanged.*

For more information on the comparisons, see [“Verifying the Database Locale” on page 1-33](#). For more information on how to handle exceptions within an ESQL program, see Chapter 8 of the [INFORMIX-ESQL/C Programmer’s Manual](#) or Chapter 4 of the [INFORMIX-ESQL/COBOL Programmer’s Manual](#).

Avoiding Partial Characters

When you use a locale that supports a multibyte code set, make sure that you define buffers large enough to avoid the generation of partial characters. Possible areas for consideration include:

- When you copy data from one buffer to another
- When you have character data that might undergo code-set conversion

For more detailed examples of partial characters, see [“Partial Characters in Column Substrings” on page 3-36](#).

Copying Character Data

When you copy data, you must ensure that the buffers are of adequate size to hold the data. If the destination buffer is not large enough for the multibyte data in the source buffer, the data might be truncated during the copy. For example, the following ESQL/C code fragment copies the multibyte data $A^1A^2A^3B^1B^2B^3$ from **buf1** to **buf2**:

```
char buf1[20], buf2[5];
:
:
stcopy("A1A2A3B1B2B3", buf1);
:
:
stcopy(buf1, buf2);
```

Because **buf2** is not large enough to hold the multibyte string, the copy truncates the string to $A^1A^2A^3B^1B^2$. To prevent this situation, ensure that the multibyte string fits into a buffer before the ESQL/C program performs the copy.

Using Code-Set Conversion

If you have a character buffer to hold character data from a database, you must ensure that this buffer is large enough to accommodate any expansion that might occur if the application uses code-set conversion. (For more information on code-set conversion, see [“Performing Code-Set Conversion” on page 1-40.](#))

If the client and database locales are different (and convertible), the application might need to expand this value. For example, if the **fname** column is defined as CHAR(8), the following ESQL/C code fragment selects an 8-byte character value into the 10-byte **buf1** host variable:

```
char buf1[10];
:
:
EXEC SQL select fname into :buf1 from tab1 where cust_num = 29;
```


You might expect that a 10-byte buffer would be adequate to hold an 8-byte character value from the database. However, if the client application expands this value to 12 bytes, the value no longer fits in the **buf1** buffer. The **fname** value would be truncated to fit into **buf1**, possibly creating partial characters if **fname** contains multibyte characters. (For more information on partial characters, see [“Partial Characters in Column Substrings” on page 3-36.](#))

To avoid this situation, make sure that you define buffers to handle the maximum character-expansion possible, 4 bytes, in the conversion between your client and database code sets.

INFORMIX-ESQL/C Features

Non-ASCII Characters in Host Variables	8-4
Locale-Sensitive Character Data Types	8-5
Extended ESQL/C Library Functions	8-7
The esqlmf Filter.	8-7
Filtering Non-ASCII Characters	8-8
Invoking the ESQL/C Filter	8-9
Handling Code-Set Conversion	8-12
Writing TEXT Values	8-13
Using the DESCRIBE Statement	8-14
The sqldata Field	8-15
The sqlname Field	8-16
Using the TRIM Function	8-16
Using TRIM with INFORMIX-SE	8-17
Using TRIM with INFORMIX-OnLine Dynamic Server	8-17
Using TRIM with INFORMIX-Universal Server	8-18

T

his chapter explains how the Global Language Support (GLS) feature affects the Informix SQL application programming interface (API) INFORMIX-ESQL/C. It contains the following topics:

- How ESQL/C handles non-ASCII characters in host variables
- How to handle the locale-sensitive data types, NCHAR and NVARCHAR, in ESQL/C applications
- Which ESQL/C library functions support locale-specific features
- How to use the ESQL/C filter, **esqlmf**
- How to handle code-set conversion from within an ESQL/C application
- How to interpret the results that the DESCRIBE statement returns for a dynamic SELECT that includes the TRIM function



***Tip:** This chapter covers GLS information that is specific to ESQL/C. For GLS information for INFORMIX-ESQL/C and INFORMIX-ESQL/COBOL, refer to [Chapter 7](#), “General SQL API Features.”*

For descriptions of ESQL/C features that are not unique to the GLS feature, see the [INFORMIX-ESQL/C Programmer's Manual](#).

Non-ASCII Characters in Host Variables

ESQL/C allows the use of non-ASCII characters in host variables when the following conditions are true:

- The client locale supports a code set with the non-ASCII characters that the host variable name contains.
You must set the client locale correctly before you preprocess and compile an ESQL/C program. For more information, see [“How Do You Set a GLS Locale?” on page 1-18](#).
- Your C compiler supports compilation of the same non-ASCII characters as the source code.
You must ensure that the C compiler supports use of non-ASCII characters in C source code. For information about how to indicate the support that your C compiler provides for non-ASCII characters, see [“Invoking the ESQL/C Filter” on page 8-9](#).

ESQL/C applications can also support non-ASCII characters in comments and SQL identifiers. For more information, see [“Non-ASCII Characters in ESQL Source Files” on page 7-3](#).

The following code fragment declares an integer host variable that contains a non-ASCII character in the host variable name and then selects a serial value into this variable:

```
/*
   This code fragment declares an integer host variable
   "hôte_ent", which contains a non-ASCII character in the
   name, and selects a serial value (code number in the
   "numéro" column of the "abonnés" table) into it.
*/

EXEC SQL BEGIN DECLARE SECTION;
    int hôte_ent;
    :
    :
EXEC SQL END DECLARE SECTION;
:
EXEC SQL select numéro into :hôte_ent from abonnés
    where nom = 'Ötker';
```

As long as the client locale supports the non-ASCII characters, you can use these characters to define indicator variables, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
      char   hôtevar[30];
      short  ind_de_hôtevar;
EXEC SQL END DECLARE SECTION;
```

You can then access indicator variables with these non-ASCII names, as the following example shows:

```
:hôtevar INDICATOR :hôtevarind

:hôtevar:hôtevarind

$hôtevar$hôtevarind
```

Locale-Sensitive Character Data Types

The SQL data types NCHAR and NVARCHAR support locale-specific data. (For more information about these data types, see [“Using Character Data Types” on page 3-11.](#))

ESQL/C supports the predefined data types **string**, **fixchar**, and **varchar** for host variables that contain character data. In addition, you can use the C **char** data type for host variables. The following table shows which of these host-variable data types you can use for NCHAR and NVARCHAR data.

Locale-Sensitive Data Type	char	fixchar	string	varchar
NCHAR	✓	✓	✓	✓
NVARCHAR	✓		✓	✓

Tip: For general information on how to use these host-variable data types with character data, see Chapter 3 of the [“INFORMIX-ESQL/C Programmer’s Manual.”](#)



Your ESQL/C program can access columns of data types NCHAR and NVARCHAR when it selects into or reads from character host variables. The following code fragment declares a **char** host variable, **hôte**, and then selects NCHAR data into the **hôte** variable:

```
/*
   This code fragment declares a char host variable "hôte",
   which contains a non-ASCII character in the name, and
   selects NCHAR data (non-ASCII names in the "nom" column
   of the "abonnés" table) into it.
*/

EXEC SQL BEGIN DECLARE SECTION;
    char hôte[10];
    :
    :
EXEC SQL END DECLARE SECTION;
:
EXEC SQL select nom into :hôte from abonnés
    where numéro > 13601;
```

When you declare ESQL/C host variables for the NCHAR and NVARCHAR data types, keep in mind the relationship between the declared size of the variable and the amount of character data that it can hold, as follows:

- If your locale supports a single-byte code set, the size of the NCHAR and NVARCHAR variable determines the number of characters that it can hold.
- If your locale supports a multibyte code set, you can no longer assume a “one-byte-per-character” relationship.

In this case, you must ensure that you declare an ESQL/C host variable large enough to accommodate the number of characters that you expect to receive from the database.

For more information, see [“The NCHAR Data Type” on page 3-11](#) and [“The NVARCHAR Data Type” on page 3-13](#).

You can insert a value that a character host variable (**char**, **fixchar**, **string**, or **vchar**) holds in columns of the NCHAR or NVARCHAR data types. For more information about how to insert host variables into the NCHAR and NVARCHAR data types, see Chapter 3 of the [INFORMIX-ESQL/C Programmer's Manual](#).

Extended ESQL/C Library Functions

ESQL/C includes the following ESQL/C library functions that provide GLS support.

DATE-Format Functions	DATETIME-Format Functions	Numeric-Format Functions	String Functions
rdatestr()	dttofmtasc()	rfmtdec()	rupshift()
rstrdate()	dtevfmtasc()	rfmtdouble()	rdownshift()
rdeffmtdate()		rfmtlong()	
rfmtdate()			

These functions accommodate features of nondefault locales such as locale-specific dates and times, non-ASCII characters, and locale-specific classification of lowercase and uppercase characters. For more information about these enhancements, see [“Enhanced ESQL Library Functions” on page 7-7](#).

The esqlmf Filter

The ESQL/C processor, **esql**, accepts C source programs that are written in the client code set (the code set of the client locale). The ESQL/C preprocessor, **esqlc**, can accept non-ASCII characters (8-bit and multibyte) in the ESQL/C source code as long as the client code set defines them.

However, the capabilities of your C compiler might limit your ability to use non-ASCII characters within an ESQL/C source file. If the C compiler does not fully support non-ASCII characters, it might not successfully compile an ESQL/C program that contains these characters. To provide support for common non-ASCII limitations of C compilers, ESQL/C provides an ESQL/C filter that is called **esqlmf**.

This section provides the following information about the ESQL/C filter:

- How the ESQL/C filter processes non-ASCII characters
- How you invoke the ESQL/C filter

Filtering Non-ASCII Characters

As part of the compilation process of an ESQL/C source program, the ESQL/C processor calls the C compiler. When you develop ESQL/C source code that contains non-ASCII characters, the way the C compiler handles such characters can affect the success of the compilation process. In particular, the following situations might affect compilation of your ESQL/C program:

- Multibyte characters might contain C-language tokens.
A component of a multibyte character might be indistinguishable from certain single-byte characters such as percent (%), comma, backslash (\), and double quote ("). If such characters exist in a quoted string, the C compiler might interpret them as C-language tokens, which can cause compilation errors or even lost characters.
- The C compiler might not be 8-bit clean.
If a code set contains non-ASCII characters (with code values that are greater than 127), the C compiler must be 8-bit clean to interpret the characters. To be 8-bit clean, a compiler must read the eighth bit as part of the code value; it must not ignore or put its own interpretation on the meaning of this eighth bit.

To filter a non-ASCII character, the ESQL/C filter converts each byte of the character to its octal equivalent. For example, suppose the multibyte character A¹A²A³ has an octal representation of \160\042\244, and appears in the **stcopy()** call.

```
stcopy("A1A2A3", dest);
```

After **esqlmf** filters the ESQL/C source file, the C compiler sees this line as follows:

```
stcopy("\160\042\244", dest); /* correct interpretation */
```

To handle the C-language-token situation, the filter prevents the C compiler from interpreting the A^2 byte (octal `\042`) as an ASCII double quote and incorrectly parsing the line as follows:

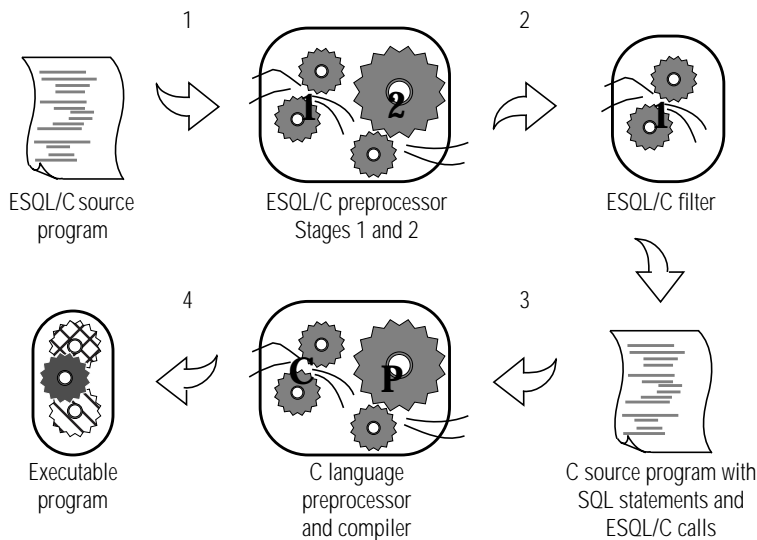
```
strcpy("A1A3, dest); /* incorrect interpretation of A2 */
```

The C compiler would generate an error for the preceding line because the line has terminated the string argument incorrectly. The **esqlmf** utility also handles the 8-bit-clean situation because it prevents the C compiler from ignoring the eighth bit of the A^3 byte. If the compiler ignores the eighth bit, it incorrectly interprets A^3 (octal `\244`) as octal `\044`.

Invoking the ESQL/C Filter

Figure 8-1 shows how an ESQL/C program that contains non-ASCII characters becomes an executable program.

Figure 8-1
Creating an ESQL/C Executable Program from a Non-ASCII Source Program



The **esql** command can automatically call the ESQL/C filter, **esqlmf**, to process non-ASCII characters. When you set the following environment variables, you tell **esql** how to invoke **esqlmf**:

- The **ESQLMF** environment variable indicates whether **esql** automatically calls the ESQL/C filter.

When you set **ESQLMF** to 1, **esql** automatically calls **esqlmf** after the ESQL/C preprocessor and before the C compiler (see [Figure 8-1 on page 8-9](#)).

- The **CC8BITLEVEL** environment variable indicates which non-ASCII characters in the ESQL/C source file that **esqlmf** filters.

Set **CC8BITLEVEL** to indicate the ability of your C compiler to process non-ASCII characters.

How **esqlmf** filters an ESQL/C source file depends on the value of the **CC8BITLEVEL** environment variable. For each value of **CC8BITLEVEL**, the following table shows the **esqlmf** command that the ESQL/C processor invokes on a ESQL/C source file.

CC8BITLEVEL Value	esqlmf Action
0	Converts all non-ASCII characters, in literal strings <i>and</i> comments, to octal constants.
1	Converts non-ASCII characters in literal strings, but not comments, to octal constants.
2	Converts non-ASCII characters in literal strings to octal constants to ensure that all the bytes in the non-ASCII characters have the eighth bit set.
3	Does not invoke esqlmf .

Important: To invoke the **esqlmf** commands that **CC8BITLEVEL** can specify, you must set the **ESQLMF** environment variable to 1.



When you set **CC8BITLEVEL** to 0, 1, or 2, the ESQL/C processor performs the following steps:

1. Converts the embedded-language statements (*source.ec*) to C-language source code (*source.c*) with the ESQL/C preprocessor
2. Filters non-ASCII characters in the preprocessed file (*source.c*) with the ESQL/C filter, **esqlmf** (if the **ESQLMF** environment variable is 1)
Before **esqlmf** begins filtering, it creates a copy of the C source file (*source.c*) that has the **.c_** file extension (*source.c_*).
3. Compiles the filtered C source file (*source.c*) with the C compiler to create an object file (*source.o*)
4. Links the object file with the ESQL/C libraries and your own libraries to create an executable program

When you set **CC8BITLEVEL** to 3, the ESQL/C processor omits step 2 in the preceding list.

If you do not set **CC8BITLEVEL**, **esql** converts non-ASCII characters in literal strings and comments. You can modify the value of **CC8BITLEVEL** to reflect the capabilities of your C compiler. For more information on **CC8BITLEVEL**, see [page 2-5](#).

Handling Code-Set Conversion

When the client and database code sets differ, the ESQL/C client application performs code-set conversion on character data. (For more information, see [“Performing Code-Set Conversion” on page 1-40.](#)) If your ESQL/C application executes in an environment in which code-set conversion might occur, make sure that the application correctly handles the following situations:

- When the application writes blobs (BYTE and TEXT values) to the OnLine Dynamic Server database, it must set the **loc_type** field in the locator structure **loc_t** to indicate the type of blob that it needs to write.

For general information on **loc_type** and **loc_t**, see Chapter 7 of the [INFORMIX-ESQL/C Programmer's Manual](#) for INFORMIX-OnLine Dynamic Server.

- When the application writes CLOB and BLOB data to the Universal Server database, it uses various LO file descriptors.

For detailed information, see Chapter 7 of the [INFORMIX-ESQL/C Programmer's Manual](#) for INFORMIX-Universal Server.

- When the application uses the **sqllda** structure to describe dynamic SQL statements, it must account for possible size differences in character data.

For general information on the **sqllda** structure, see Chapter 10 of the [INFORMIX-ESQL/C Programmer's Manual](#).

- When the application has character data that might undergo code-set conversion, you must declare character buffers that can hold the data.

For more information, see [“Avoiding Partial Characters” on page 7-25.](#)

Writing TEXT Values



Important: The following information pertains to INFORMIX-OnLine Dynamic Server only. For information on how to write TEXT values in INFORMIX-Universal Server, refer to the “[INFORMIX-ESQL/C Programmer’s Manual](#).”

ESQL/C uses the **loc_t** locator structure to read blobs from and write blobs to the database server. The **loc_type** field of this structure indicates the data type of the blob value that the structure describes. When the client and database code sets are the same (no code-set conversion), the client application does not need to set the **loc_type** field explicitly because the database server can determine the blob data type implicitly. The database server assumes that character data has the TEXT data type and noncharacter data has the BYTE data type.

However, if the client and database code sets are different and convertible, the client application must know the data type of this blob in order to determine whether to perform code-set conversion on the blob data. Before your ESQL/C client application inserts a blob into the database, it must explicitly set the **loc_type** field of the blob as follows:

- For a TEXT blob, the ESQL/C client application must set the **loc_type** field to SQLTEXT before the INSERT statement.

The client performs code-set conversion on TEXT blob data before it sends this data to the database for insertion.

- For a BYTE blob, the ESQL/C client application must set the **loc_type** field to SQLBYTES before the INSERT statement.

The client does *not* perform code-set conversion on BYTE blob data before it sends this data to the database for insertion.



Important: The **sqltypes.h** header file defines the data-type constants SQLTEXT and SQLBYTES. You must include this header file in your ESQL/C source file to use these constants. For more information on the **sqltypes.h** header file, see Chapter 2 of the “[INFORMIX-ESQL/C Programmer’s Manual](#).”

Your ESQL/C source code does not need to set **loc_type** before it reads blob data from a database. The database server obtains the data type of the blob from the database and sends this data type to the client with the data.

If you set **loc_bufsize** to -1, ESQL/C allocates memory to hold a single blob value. It stores the address of this memory buffer in the **loc_buffer** field of the **loc_t** structure. If the client application performs code-set conversion on TEXT blobs that the database server retrieves, ESQL/C handles any possible data expansion as follows:

1. Frees the existing memory that the **loc_buffer** field references
2. Reallocates a memory buffer that is large enough to store the expanded TEXT data
3. Assigns the address of this new memory buffer to the **loc_buffer** field
4. Assigns the size of the new memory buffer to the **loc_bufsize** field

If this reallocation occurs, ESQL/C changes the memory address at which it stores the TEXT blob. If your ESQL/C program references this address, the program must account for the address change.

ESQL/C does not need to reallocate memory for the TEXT data if code-set conversion does not expand the TEXT data or if it condenses the data. In either of these cases, the **loc_buffer** field remains unchanged, and the **loc_bufsize** field contains the size of the buffer that the **loc_buffer** field references.

Using the DESCRIBE Statement

The **sqllda** structure is a dynamic-management structure that contains information about columns in dynamic SQL statements. The DESCRIBE...INTO statement uses the **sqllda** structure to return information about the select-list columns of a SELECT statement. It sets the **sqlvar** field of an **sqllda** structure to point to a sequence of partially filled **sqlvar_struct** structures. Each structure describes a single select-list column. (For more information about how to determine select-list columns, see “Using an sqllda Structure” in Chapter 10 of the [INFORMIX-ESQL/C Programmer's Manual](#).)

Each **sqlvar_struct** structure contains character data for the column name and the column data. When the ESQL/C client application fills this structure, the column name and the column data are in the client code set. When the database server fills this structure (when it executes a DESCRIBE...INTO statement), this character data is in the database code set.

When the client application performs code-set conversion between the client and database code sets, the number of bytes that is required to store the column name and column data in the client code set might not equal the number that is required to store this same information in the database code set. Therefore, the size of the character data in **sqlvar_struct** might increase or decrease during code-set conversion. To handle this possible difference in size, the client application must ensure that it correctly handles the character data in the **sqlvar_struct** structure.

The sqldata Field

To hold the column data, the client application must allocate a buffer and set **sqldata** to point to this buffer. If your client application might perform code-set conversion, it must allocate sufficient storage to handle the increase in the size of the column data that might occur.

When the DESCRIBE...INTO statement sets the **sqllen** field, the **sqllen** value indicates the length of the column data in the database code set. Therefore, if you use the value of **sqllen** that the DESCRIBE...INTO statement retrieves, you might not allocate a buffer that is sufficiently large for the data when it is in the client code set. For example, the following code fragment allocates an **sqldata** buffer with the **malloc()** system call:

```
EXEC SQL include sqlda;
:
:
struct sqlda *q_desc;
:
:
EXEC SQL describe sqlstmt_id into q_desc;
:
:
q_desc->sqlvar[0].sqldata = (char *)malloc(q_desc->sqlvar[0].sqllen);
```

In the preceding code fragment, the client application might truncate characters that it converts because the client application uses the **sqllen** value to determine the buffer size. Instead, increase the buffer to four times its original size when you allocate a buffer, as shown in the following code fragment:

```
EXEC SQL include sqlda;
EXEC SQL define BUFSIZE_FACT 4;
:
:
struct sqlda *q_desc;
:
:
q_desc->sqlvar[0].sqllen = q_desc->sqlvar[0].sqllen * BUFSIZE_FACT + 1;
q_desc->sqlvar[0].sqldata = (char *)malloc(q_desc->sqlvar[0].sqllen);
```

Informix suggests a buffer-size factor (**BUFSIZE_FACT**) of 4 because a multibyte character has a maximum size of 4 bytes.

The sqlname Field

The **sqlname** field contains the name of the column. When the client application performs code-set conversion, this column name might also undergo expansion when the application converts it from the database code set to the client code set. Because the ESQL/C application stores the buffer for **sqlname** data in its internal work area, your ESQL/C source code does not have to handle possible buffer-size increases. Your code processes the contents of **sqlname** in the client code set.

Using the TRIM Function

When you dynamically execute a SELECT statement, the DESCRIBE statement can return information about the select-list columns at runtime. DESCRIBE returns the data type of a select-list column into the appropriate field of the dynamic-management structure that you use. (For more information, see “Determining the Data Type of a Column” in Chapter 10 of the [INFORMIX-ESQL/C Programmer's Manual](#).)

When you use the DESCRIBE statement on a prepared SELECT statement with the TRIM function in its select list, the data type of the trimmed column that DESCRIBE returns depends on the database server that you use and the data type of the column to be trimmed (the *source character-value expression*). For more information on the source character-value expression, see the description of the TRIM function in the [Informix Guide to SQL: Syntax](#).

Using TRIM with INFORMIX-SE

For INFORMIX-SE, the DESCRIBE statement returns the same data type as the source character-value expression. When you dynamically execute the following SELECT statement with an SE database server, the DESCRIBE statement returns a data type of SQLCHAR, the same data type as the select-list column, **manu_code** (the source character-value expression).

```
SELECT TRIM(manu_code) FROM manufact;
```

Using TRIM with INFORMIX-OnLine Dynamic Server

For INFORMIX-OnLine Dynamic Server, the data type that the DESCRIBE statement returns depends on the data type of the source character-value expression, as follows:

- If the source character-value expression is data type CHAR or VARCHAR, DESCRIBE returns the data type of the trimmed column as SQLVCHAR.
- If the source character-value expression is data type NCHAR or NVARCHAR, DESCRIBE returns the data type of the trimmed column as SQLNVCHAR.

The following SELECT statement contains the **manu_code** column, which is defined as a CHAR data type, and the **cat_advert** column, which is defined as a VARCHAR column. When you describe the following SELECT statement with an OnLine database server, DESCRIBE returns a data type of SQLVCHAR for both trimmed columns:

```
SELECT TRIM(manu_code), TRIM(cat_advert) FROM catalog;
```

If the **manu_code** column is defined as NCHAR instead, DESCRIBE returns a data type of SQLNVCHAR for this trimmed column.



Important: The **sqltypes.h** header file defines the data-type constants **SQLCHAR**, **SQLVCHAR**, and **SQLNVCHAR**. You must include this header file in your **ESQL/C** source file in order to use these constants. For more information on the **sqltypes.h** header file, see Chapter 2 of the “[INFORMIX-ESQL/C Programmer's Manual](#).”

Using TRIM with INFORMIX-Universal Server

For INFORMIX-Universal Server, the data type that the **DESCRIBE** statement returns depends on the data type of the source character-value expression, as follows:

- If the source character-value expression is data type **CHAR** or **VARCHAR**, **DESCRIBE** returns the data type of the trimmed column as **SQLVCHAR**.
- If the source character-value expression is data type **NCHAR** or **NVARCHAR**, **DESCRIBE** returns the data type of the trimmed column as **SQLNVCHAR**.



Important: **TRIM** does not support the **LVARCHAR** data type in Universal Server.

The following **SELECT** statement contains the **manu_code** column, which is defined as a **CHAR** data type, and the **cat_advert** column, which is defined as a **VARCHAR** column. When you describe the following **SELECT** statement with a database server, **DESCRIBE** returns a data type of **SQLVCHAR** for both trimmed columns:

```
SELECT TRIM(manu_code), TRIM(cat_advert) FROM catalog;
```

If the **manu_code** column is defined as **NCHAR** instead, **DESCRIBE** returns a data type of **SQLNVCHAR** for this trimmed column.



Important: The **sqltypes.h** header file defines the data-type constants **SQLCHAR**, **SQLVCHAR**, and **SQLNVCHAR**. You must include this header file in your **ESQL/C** source file in order to use these constants. For more information on the **sqltypes.h** header file, see Chapter 2 of the “[INFORMIX-ESQL/C Programmer's Manual](#).”

Managing GLS Files

UNIX

This appendix provides the following information about how to manage GLS files:

- How GLS files provide locale-related information
- What information a GLS locale file provides
- What other GLS files Informix products use
- How to determine which GLS files you can remove if you do not use them
- How to list the available GLS locales and code-set conversions with the **glfiles** utility ♦

Accessing GLS Files

Informix products access the following GLS files to obtain locale-related information.

GLS Files	Reference
GLS locale files	page A-3
Code-set-conversion files	page A-14
Code-set files	page A-17
The registry file	page A-18

In general, you do not need to examine the GLS files. However, you might want to look at these files to determine the following locale-specific information.

Locale-Specific Information	GLS File to Examine	Reference
Collation order		
Exact localized order	Source locale file (*.lc): LC_COLLATION category	Figure A-1 on page A-12
Exact code-set collation order	Source code-set file (*.cm)	page A-17
Character mappings		
Locale-specific mapping between uppercase and lowercase characters	Source locale file (*.lc): LC_CTYPE category	Figure A-1 on page A-12
Locale-specific classification of characters	Source locale file (*.lc): LC_CTYPE category	Figure A-1 on page A-12
Code-set-specific character mappings	Source code-set file (*.cm)	page A-17
Mappings between characters of the source and target code sets	Source code-set-conversion file (*.cv)	page A-14
Method for character mismatches during code-set conversion	Source code-set-conversion file (*.cv)	page A-14
Code points for characters	Source code-set file (*.cm)	page A-17

(1 of 2)

Locale-Specific Information	GLS File to Examine	Reference
End-user formats		
Numeric (non-monetary) data	Source locale file (*.lc): LC_NUMERIC category	Figure A-1 on page A-12
Monetary data	Source locale file (*.lc): LC_MONETARY category	Figure A-1 on page A-12
Date data	Source locale file (*.lc): LC_TIME category	Figure A-1 on page A-12
Time data	Source locale file (*.lc): LC_TIME category	Figure A-1 on page A-12

(2 of 2)

GLS Locale Files

The *locale file* defines a GLS locale. It describes the basic language and cultural conventions that are relevant to the processing of data for a given language and territory. This section provides the following information about GLS locale files:

- How the locale file groups locale-specific information into locale categories
- Where locale files reside and how these locale files are named

Locale Categories

A GLS locale file groups locale-specific information into the six locale categories that the following table shows.

Locale Category	Description
CTYPE	Controls the behavior of character classification and case conversion.
COLLATION	Controls the behavior of string comparisons.
NUMERIC	Controls the behavior of non-monetary numeric end-user formats.
MONETARY	Controls the behavior of currency end-user formats.
TIME	Controls the behavior of date and time end-user formats.
MESSAGES	Controls the definitions of affirmative and negative responses to messages.

The CTYPE and COLLATION categories primarily affect how the database server stores and retrieves character data in a database. The NUMERIC, MONETARY, and TIME categories affect how a client application formats the internal values of the associated SQL data types.

The default locale, U.S. English, provides the following behavior for each of the locale categories.

Locale Category	In Default Locale (U.S. English)
CTYPE	The default code set classifies characters.
	The default code set is ISO8859-1. ♦
	The default code set is Microsoft 1252. ♦
COLLATE	The default locale does not define a localized order. Therefore, the database server collates NCHAR and NVARCHAR data in code-set order.

(1 of 2)

UNIX

Windows

Locale Category	In Default Locale (U.S. English)
NUMERIC	<p>The following numeric notation for use in numeric end-user formats:</p> <ul style="list-style-type: none">■ Thousands separator: comma (,)■ Decimal separator: period (.)■ Number of digits between thousands separators: 3■ Symbol for positive number: plus (+)■ Symbol for negative number: minus (-)■ No alternative digits for era-based dates
MONETARY	<p>The following currency notation for use in monetary end-user formats:</p> <ul style="list-style-type: none">■ Currency symbol: dollar sign (\$) appears before the currency value■ Thousands separator: comma (,)■ Decimal separator: period (.)■ Number of digits between thousands separators: 3■ Symbol for positive number: plus (+)■ Symbol for negative number: minus (-) <p>Default scale for MONEY columns: 2</p>
TIME	<p>The following date and time end-user formats:</p> <ul style="list-style-type: none">■ DATE values: %m/%d/%iy■ DATETIME values: %iY-%m-%d %H:%M:%S <p>No definitions for era-based dates.</p>
MESSAGES	None

(2 of 2)

The following sections describe each of these locale categories in more detail.

NLS

The CTYPE Category

The CTYPE category defines how to classify the characters of the code set that the locale supports. This category includes specifications for which characters the locale classifies as spaces, blanks, control characters, digits, uppercase letters, lowercase letters, and punctuation. This category might also include mappings between uppercase and lowercase letters. Informix products access this category when they need to determine the validity of an identifier name, shift the case of a character, or compare characters.

For Version 7.2 and later products, the CTYPE locale category replaces the functionality of the NLS environment variable, `LC_CTYPE`. ♦

The COLLATION Category

The COLLATION category defines the localized order. When an Informix product needs to compare two strings, it first breaks up the strings into a series of collation elements. The database server compares each pair of collation elements according to the collation weights of each element. The COLLATION category provides support for the following capabilities:

- Multicharacter collation elements define characters that the database server should collate as a single unit.

For example, the localized order might treat the Spanish double-`l` (`ll`) as a single collation element instead of a pair of `l`'s.

- Equivalence classes assign the same collation weight to different collation elements.

For example, the localized order might specify that `a` and `A` are an equivalence class (`a` and `A` are equivalent characters).

The difference in collation order is the only distinction between the CHAR and NCHAR data types and the VARCHAR and NVARCHAR data types. For more information, see [“Using Character Data Types” on page 3-11](#).

If a locale does not contain a COLLATION category, Informix products use code-set order for collation of all character data types (CHAR, NCHAR, NVARCHAR, TEXT, and VARCHAR).

NLS

For Version 7.2 and later products, the COLLATION locale category replaces the functionality of the NLS environment variable, `LC_COLLATE`. ♦



NLS

The NUMERIC Category

The NUMERIC category defines the following numeric notation for end-user formats of numeric, non-monetary values:

- The numeric decimal separator
- The numeric thousands separator
- The number of digits to group between each appearance of a non-monetary thousands separator
- The characters that indicate positive and negative numbers

This numeric notation applies to the end-user formats of data for numeric (DECIMAL, INTEGER, SMALLINT, FLOAT, SMALLFLOAT) columns within a client application. For more information on end-user formats, see [page 1-12](#).

Important: Information in the NUMERIC category does not affect the internal format of the numeric data types in the database.

The NUMERIC category also defines alternative digits for use in era-based dates and times. For more information on alternative digits, see “[Alternative Date Formats](#)” on [page 2-29](#) and “[Alternative Time Formats](#)” on [page 2-37](#).

For Version 7.2 and later products, the NUMERIC locale category replaces the functionality of the NLS environment variable, LC_NUMERIC. ♦

The MONETARY Category

The MONETARY category defines the following currency notation for end-user formats of monetary values:

- The currency symbol and whether it appears before or after a monetary value
- The monetary decimal separator
- The monetary thousands separator
- The number of digits to group between each appearance of a monetary thousands separator
- The characters that indicate positive and negative monetary values and the position of (before or after) these characters
- The number of fractional digits (those to the right of the decimal point) to display



NLS

This currency notation applies to the end-user formats of data from MONEY columns within a client application. For more information on end-user formats, see [page 1-12](#).

Important: *Information in the MONETARY category does not affect the internal format of the MONEY data type in the database.*

The MONETARY category also defines the default scale for a MONEY column. For the default locale (U.S. English), the database server stores the data type MONEY(*precision*) in the same internal format as the data type DECIMAL(*precision*,2). A nondefault locale can define a different default scale. For more information on default scales, see “[Default Values for the Scale Parameter](#)” on [page 3-48](#).

To customize the end-user format of monetary values, use the DBMONEY environment variable. For more information, see “[Customizing Monetary Values](#)” on [page 1-53](#).

For Version 7.2 and later products, the MONETARY locale category replaces the functionality of the NLS environment variable, LC_MONETARY. ♦

The TIME Category

The TIME category lists characters and symbols that format date and time values. This information includes the names and abbreviations for days of the week and months of the year. It also includes special representations for dates, time (12-hour and 24-hour), and DATETIME values.

These representations can include the names of eras (as in the Japanese Imperial era system) and non-Gregorian calendars (such as the Arabic lunar calendar). The locale determines what calendar to use (Gregorian, Hebrew, Arabic, Japanese Imperial, and so on) when it scans or prints a month, day, or year. For more information on the scan and print operations, see “[End-User Formats](#)” on [page 1-12](#).

If the locale supports era-based dates and times, the TIME category defines the full and abbreviated era names and special date and time representations. For more information, see “[Alternative Date Formats](#)” on [page 2-29](#) and “[Alternative Time Formats](#)” on [page 2-37](#).



NLS

This date and time information applies to the end-user formats of data in DATE and DATETIME columns within a client application. For example, the ESQL/COBOL library routine ECO-DAT uses the date end-user format in the TIME category of the client locale to determine how to print a date string from an internal DATE value.

Important: Information in the TIME category does not affect the internal format of the DATE and DATETIME data types in the database.

To customize the format of date and time values, use the GL_DATE and GL_DATETIME environment variables. For more information, see [“Customizing Date and Time End-User Formats” on page 1-50](#).

For Version 7.2 and later products, the TIME locale category replaces the functionality of the NLS environment variable, LC_TIME. ♦

The MESSAGES Category

The MESSAGES category defines the format for affirmative and negative responses. This category is optional. Informix products do not use the strings that the MESSAGES category defines.

To obtain the locale name for the MESSAGES category of the client locale, a client application uses the locale that CLIENT_LOCALE indicates. If CLIENT_LOCALE is not set, the client sets the category to the default locale.

Location of Locale Files

When an Informix product needs to obtain locale-specific information, it accesses one of the following GLS locale files, depending on the platform:

- GLS locale files reside in the following file:

`$INFORMIXDIR/gls/lcX/lg_tr/codemodf.lco` ♦

- GLS locale files reside in the following file:

`%INFORMIXDIR%\gls\lcX\lg_tr\codemodf.lco` ♦

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the Informix product, and **gls** is the subdirectory that contains the GLS files.

UNIX

Windows

This section describes the remaining elements in the pathname of GLS locale files, as follows:

- For information on the **lcX** subdirectory and its *lg_tr* locale-file subdirectories, see “Locale-File Subdirectories”.
- For information on the locale object (*.lco) and source (*.lc) files, see “[Locale Source and Object Files](#)” on page A-11.
- For information on the name of the *codemodf.lco* file, see “[Locale Filenames](#)” on page A-12.

Locale-File Subdirectories

The subdirectories of the **lcX** subdirectory, where *X* represents the version number for the locale object-file format, contain the GLS locale files. These subdirectories have names of the form *lg_tr*, where *lg* is the 2-character language name and *tr* is the 2-character territory name that the locale supports.

UNIX

The GLS locale-file subdirectories are in the **\$INFORMIXDIR/gls/lcX** directory. ♦

Windows

The GLS locale-file subdirectories are in the **%INFORMIXDIR%\glslcX** directory. ♦

The following table shows some languages and territories that Informix products can support, along with their associated locale-file subdirectory names.

Language	Territory	Locale-File Subdirectory
English	United States	en_us
	Great Britain	en_gb
	Australia	en_au
German	Germany	de_de
	Austria	de_at
	Switzerland	de_ch
French	Belgium	fr_be
	Canada	fr_ca
	Switzerland	fr_ch
	France	fr_fr

NLS

Version 7.2 and later products also provide GLS locales that are compatible with many operating-system locales. These locale files reside in the **os** subdirectory of the **lcX** subdirectory in your Informix installation. The **os** subdirectory contains a subdirectory for each operating-system locale that the platform of your Informix product supports.

UNIX

The GLS operating-system locales are in subdirectories of the **\$INFORMIXDIR/gls/lcX/os** directory. ♦

Windows

The GLS operating-system locales are in subdirectories of the **%INFORMIXDIR%\glslcX\os** directory. ♦

Informix provides these operating-system GLS locale files for backward compatibility with Informix NLS products. For more information, see the [Informix Migration Guide](#). ♦

Locale Source and Object Files

Each locale file has the following two forms:

- A locale *source* file is an ASCII file that defines the locale categories for the locale.
This file has the **.lc** file extension and serves as documentation for the corresponding object file.
- A locale *object* file is a compiled form of the locale information.
Informix products use the object file to obtain locale information quickly. Locale object files have the **.lco** file extension.

The header of the locale source file (.lc) lists the language, territory, code set, and any optional locale modifier of the associated locale. A section of the locale source file supports each of the locale categories, as Figure A-1 shows.

Figure A-1
Locale Categories in a Locale Source File

Locale Category	Locale-File Category	Reference
CTYPE	LC_CTYPE	page A-6
COLLATE	LC_COLLATE	page A-6
NUMERIC	LC_NUMERIC	page A-7
MONETARY	LC_MONETARY	page A-7
TIME	LC_TIME	page A-8
MESSAGES	LC_MESSAGES	page A-9

Locale Filenames

To conform to DOS 8.3 naming conventions, a GLS locale file uses a condensed form of the code-set name, *codemodf*, in its filenames. The 4-character *code* name of each locale file is the hexadecimal representation of the code-set number for the code set that the locale supports. The 4-character *modf* name is the optional locale modifier.

For example, the ISO8859-1 code set has an IBM CCSID number of 819 in decimal and 0333 in hexadecimal. Therefore, the 4-character name of a locale source file that supports the ISO8859-1 code set is **0333.lc**. The following table shows some code sets and locale modifiers that Informix products can support, along with their associated locale source filenames.

Code Set	Locale Modifier	Locale Source File
ISO8859-1 (IBM CCSID 819)	<i>none</i>	0333.lc
	Dictionary	0333dict.lc
Microsoft 1252 (West Europe)	<i>none</i>	04e4.lc
	Dictionary	04e4dict.lc
IBM CCSID 850	<i>none</i>	0352.lc
	Dictionary	0352dict.lc

A French locale that supports the ISO8859-1 code set has a GLS locale that is called **0333.lc** file in the **fr_fr** locale-file subdirectory. The default locale, U.S. English, also uses the ISO8859-1 code set (on UNIX platforms); a locale file that is called **0333.lc** is also in the **en_us** locale-file subdirectory. Because both the French and U.S. English locales support the Microsoft 1252 code set, both the **fr_fr** and **en_us** locale-file subdirectories contain a **04e4.lc** locale file as well.

Figure A-2 on page A-23 shows the output that the **glfiles** utility generates for locale files. ♦

Tip: To save disk space, you might want to keep only those locale files that you intend to use. For information on which locale files you can remove, refer to “[Removing Locale and Code-Set-Conversion Files](#)” on page A-20.

UNIX



Other GLS Files

In addition to GLS locale files, Informix products might also use the following GLS files:

- Code-set-conversion files map one code set to another.
- Code-set files define code-point values for code sets.
- The Informix **registry** file converts locale aliases to valid locale files.

This section describes each of these types of GLS files. For information on GLS locale files, see [page A-3](#).

Code-Set-Conversion Files

The *code-set-conversion file* describes how to map each character in a particular source code set to the characters of a particular target code set. Informix products can perform a given code-set conversion if code-set-conversion files exist to describe the mapping between the two code sets.

Important: *A client application checks the code sets that the client and database locales support when it begins execution. If code sets are different, and no code-set-conversion files exist, the client application generates an error. For information, see “Establishing a Database Connection” on page 1-31.*

When an Informix product needs to obtain code-set-conversion information, it accesses one of the following GLS code-set-conversion files, depending on the platform:



UNIX

Windows

- GLS code-set-conversion files reside in the following file:
`$INFORMIXDIR/gls/cvY/code1code2.cvo` ♦
- GLS code-set-conversion files reside in the following file:
`%INFORMIXDIR%\gls\cvY\code1code2.cvo` ♦

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the Informix product, **gls** is the subdirectory that contains the GLS files, and **Y** represents the version number for the code-set-conversion object-file format.

This section describes the remaining elements in the pathname of GLS code-set-conversion files, as follows:

- For information on the code-set-conversion object (*.cvo) and source (*.cv) files, see “Code-Set-Conversion Source and Object Files” below.
- For information on the name of the *code1code2.cvo* file, see [page A-16](#).
- For information about which code-set-conversion files an Informix products must locate, see [page A-16](#).

Code-Set-Conversion Source and Object Files

Each code-set-conversion file has the following two forms:

- The code-set-conversion *source* file is an ASCII file that describes the mapping to use for one direction of the code-set conversion.
This file has a .cv extension and serves as documentation for the corresponding object file.
- The code-set-conversion *object* file is a compiled form of the code-set-conversion information.
Informix products use the object file to obtain code-set-conversion information quickly. Object code-set-conversion files have a .cvo file extension.

The header of the code-set-conversion source file (.cv) lists the two code sets that it converts, and the direction of the conversion.

Code-Set-Conversion Filenames

To conform to DOS 8.3 naming conventions, GLS code-set-conversion files use a condensed form of the code-set names, *code1code2*, in their filenames. The 8-character name of each code-set-conversion file is derived from the hexadecimal representation of the code-set numbers of the source code set (*code1*) and the target code set (*code2*).

For example, the ISO8859-1 code set has an IBM CCSID number of 819 in decimal and 0333 in hexadecimal. The IBM CCSID 437 code set, a common IBM UNIX code set, has a hexadecimal value of 01b5. Therefore, the **033301b5.cv** code-set-conversion file describes the conversion from the CCSID 819 code set to the CCSID 437 code set.

Required for Code-Set Conversion

Informix products use the Informix Code-Set Name-Mapping file to translate between code-set names and the more compact code-set numbers. You can use the **registry** file to find the hexadecimal values that correspond to code-set names or code-set numbers. For more information, see [“The Informix registry File” on page A-18](#).

Most code-set conversion requires *two* code-set-conversion files. One file supports conversion of characters in code set A to their counterparts in code set B. Another supports the conversion in the return direction (from B to A). Such conversions are called *two-way* code-set conversions. For example, the code-set conversion between the CCSID 437 code set (hexadecimal 01b5 code number) and the CCSID 819 (or ISO8859-1 with a hexadecimal 0333 code number) code set requires the following two code-set-conversion files:

- The **01b50333.cv** file describes the mappings to use when Informix products convert characters in the CCSID 437 code set to those in the ISO8859-1 code set.
- The **033301b5.cv** file describes the mappings to use when Informix products convert characters in the ISO8859-1 code set to those in the CCSID 437 code set.

To be able to convert between these two code sets, an Informix product must be able to locate *both* these code-set-conversion object files. Performing the conversion on only one direction would result in mismatched characters. For more information on mismatched characters, see [page 1-42](#).

The following table shows some of the code-set conversions that Informix products can support, along with their associated code-set-conversion source filenames.

Source Code Set	Target Code Set	Code-Set-Conversion Source File
ISO8859-1	Microsoft 1252	033304e4.cvo
Microsoft 1252	ISO8859-1	04e40333.cvo
ISO8859-1	IBM CCSID 850	03330352.cvo
IBM CCSID 850	ISO8859-1	03520333.cvo
Microsoft 1252	IBM CCSID 850	04e40352.cvo
IBM CCSID 850	Microsoft 1252	035204e4.cvo

UNIX



[Figure A-4 on page A-25](#) shows the output that the **glfiles** utility generates for these code-set-conversion files. ♦

Tip: To save disk space, you might want to keep only those code-set-conversion files that you intend to use. For information about which code-set-conversion files you can remove, refer to [“Removing Locale and Code-Set-Conversion Files” on page A-20](#)

Code-Set Files

An Informix *code-set file* (also called a *charmap* file) defines a code set for subsequent use by locale and code-set-conversion files. A GLS locale includes the appropriate code-set file for the code set it supports. In addition, Informix products can perform code-set conversion between the code sets that have code-set files.

UNIX

Windows

When an Informix product needs to obtain code-set information, it accesses one of the following GLS code-set files, depending on the platform:

- GLS code-set files reside in the following file:

`$INFORMIXDIR/gls/cmZ/code.cmo` ♦

- GLS code-set files reside in the following file:

`%INFORMIXDIR%\gls\cvY\code.cmo` ♦

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the Informix product, **gls** is the subdirectory that contains the GLS files, and **Z** represents the version number for the code-set object-file format.

Each code-set file has the following two forms:

- The code-set *source* file is an ASCII file that describes the characters of a character set.

This file has a **.cm** extension and serves as documentation for the corresponding object file.

- The code-set *object* file is a compiled form of the code-set information.

The object file is used to create locale object files. Object code-set files have a **.cmo** file extension.

[Figure A-5 on page A-26](#) shows the output that the **glfiles** utility generates for code-set files. ♦

UNIX

The Informix registry File

The Informix Code-Set Name-Mapping file, which is called **registry**, is an ASCII file that associates code-set names and aliases with their code-set numbers. A code-set number is based on the IBM CCSID numbering scheme. Informix products use code-set numbers to determine the filenames of locale and code-set-conversion files. For more information on locale files and code-set-conversion files, see [page A-12](#) and [page A-16](#), respectively.

For example, you can specify the French locale that supports the ISO8859-1 code set with any of the following locale names as locale aliases:

- The full code-set name
`fr_fr.8859-1`
- The decimal value of the IBM CCSID number
`fr_fr.819`
- The hexadecimal value of the IBM CCSID number
`fr_fr.0333`

When you specify a locale name with either of the first two forms, Informix products use the Informix Code-Set Name-Mapping file to translate between code-set names (8859-1) or code-set number (819) to the condensed code-set name (0333). For information about the file format and search algorithm that Informix products use to convert code-set names to code-set numbers, refer to the comments at the top of the **registry** file.

When an Informix product needs to obtain information about locale aliases, it accesses one of the following GLS code-set files, depending on the platform:

UNIX

- Informix **registry** file has the following path:

`$INFORMIXDIR/gls/cmZ/registry` ♦

Windows

- Informix **registry** file has the following path:

`%INFORMIXDIR%\gls\cv\registry` ♦

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the Informix product, **gls** is the subdirectory that contains the GLS files, and **Z** represents the version number for the code-set object-file format.

Warning: Do not remove the Informix Code-Set Name-Mapping file, **registry**, from the Informix directory. Do not modify this file. Informix products use this file for the language processing of all locales.



Removing Unused Files

An Informix product contains the following GLS files:

- Locale files: source (*.lc) and object (*.lco)
- Code-set-conversion files: source (*.cv) and object (*.cvo)
- Code-set files: source only (*.cm)

To save disk space, you might want to keep only those files that you intend to use. This section describes which of these files you can safely remove from your Informix installation.

Removing Locale and Code-Set-Conversion Files

You can safely remove the following GLS files from your Informix installation:

- *For those locales that you do not intend to use*, you can remove locale source and object files (.lc and .lco) from the subdirectories of the lcX subdirectory in your Informix installation.

For more information on the lcX pathname, see [“Locale-File Subdirectories” on page A-10](#).

- *For those code-set conversions that you do not intend to use*, you can remove code-set-conversion source and object files (.cv and .cvo) from the subdirectories of the cvY subdirectory in your Informix installation.

For more information on the cvY pathname, see [“Code-Set-Conversion Filenames” on page A-16](#).



Warning: Do not remove the locale object file for the U.S. 8859-1 English locale, **0333.lco** in the **en_us** locale-file subdirectory. In addition, do not remove the Informix Code-Set Name-Mapping file, **registry**. Informix products use these files for the language processing of all locales.

Because Informix products do not access source versions of locale and code-set conversion files, you can safely remove them. However, these files do provide useful on-line documentation for the supported locales and code-set conversions. If you have enough disk space, Informix recommends that you keep these source files for the GLS locales (*.lc) and code-set conversions (*.cv) that your Informix installation supports.

Removing Code-Set Files

Informix provides the source version of code-set files (.cm) as on-line documentation for the locales and code-set conversions that use them. Because Informix products do not access source code-set files, you can safely remove them. However, if you have enough disk space, Informix recommends that you keep these source files for the GLS locales and code-set conversions that your Informix installation supports.

UNIX

The glfiles Utility

To comply with DOS 8.3 naming conventions, Informix products use condensed filenames to store GLS locales and code-set-conversion files. These filenames do not match the names of the locales and code sets that the end user uses. You can use the **glfiles** utility to generate a more readable list of the following GLS-related files:

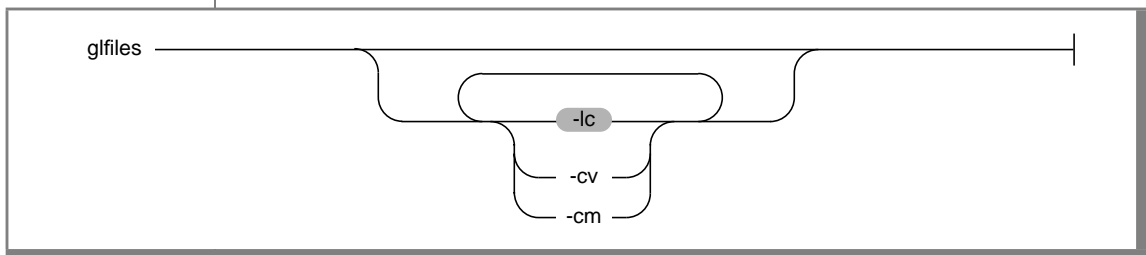
- The GLS locales that are available on your system
- The Informix code-set-conversion files that are available on your system
- The Informix code-set files that are available on your system

Before you run **glfiles**, take the following steps:

- Set the **INFORMIXDIR** environment variable to the directory in which you install your Informix product.
If you do not set **INFORMIXDIR**, **glfiles** checks the **/usr/informix** directory for the GLS files.
- Change directories to the directory where you want the files that **glfiles** generates to reside.

The utility creates the GLS file listings in the current directory.

The following diagram shows the syntax of the **glfiles** utility.



Listing GLS Locale Files

The **glfiles** utility can create a file that lists the available GLS locales in the following ways:

- When you specify the **-lc** command-line option
- When you omit all command-line options

For each **lcX** subdirectory in **\$INFORMIXDIR/gls**, **glfiles** creates a file in the current directory that is called **lcX.txt**, where **X** is the version number of the locale object-file format. The **lcX.txt** file lists the locales in alphabetical order, sorted on the name of the GLS locale object file.

Figure A-2 shows a sample file, **lc11.txt**, that contains the available GLS locales.

```
Filename: lc11/ar_ae/0441.lco
Language: Arabic
Territory: United Arabic Emirates
Modifier: greg
Code Set: 8859-6
Locale Name: ar_ae.8859-6

Filename: lc11/ar_ae/0441greg.lco
Language: Arabic
Territory: United Arabic Emirates
Modifier: greg
Code Set: 8859-6
Locale Name: ar_ae.8859-6
:
:
Filename: lc11/en_us/0333.lco
Language: English
Territory: United States
Code Set: 8859-1
Locale Name: en_us.8859-1

Filename: lc11/en_us/0333dict.lco
Language: English
Territory: United States
Modifier: dict
Code Set: 8859-1
Locale Name: en_us.8859-1

Filename: lc11/en_us/0352.lco
Language: English
Territory: United States
Code Set: PC-Latin-1
Locale Name: en_us.PC-Latin-1

Filename: lc11/en_us/04e4.lco
Language: English
Territory: United States
Code Set: CP1252
Locale Name: en_us.CP1252
:
:
```

Figure A-2
*Sample glfiles File
for GLS Locales*

NLS

The **lcX.txt** file also contains GLS locales that are compatible with many operating-system locales. Figure A-3 shows the format in **lc11.txt** for these GLS operating-system locales. For more information, see [“Locale-File Subdirectories”](#) on page A-10.

```
*****
*
* The following locales are provided to ensure that
* Informix servers, starting with version 7.20, can
* open databases created by version 6.xx and 7.1x
* Informix servers. Because these locales are based
* on operating system locales and operating system
* locales do not provide the required informatoin,
* no Language, territory, Modifier, or Code Set
* names are provided below.
*
*****

Filename: lc11/os/de
Locale Name: de

Filename: lc11/os/en_US
Locale Name: en_US

Filename: lc11/os/es
Locale Name: es

Filename: lc11/os/fr
Locale Name: fr

Filename: lc11/os/iso_8859_1
Locale Name: iso_8859_1
:
:
```

Figure A-3
*Format for
Operating-System
Locales*



Examine the **lcX.txt** file(s) to determine the GLS locales that the **\$INFORMIXDIR/gls/lcX** directory on your system supports.

Listing Code-Set-Conversion Files

When you specify the **-cv** command-line option, the **glfiles** utility creates a file that lists the available code-set-conversion files. For each **cvY** subdirectory in **\$INFORMIXDIR/gls**, **glfiles** creates a file in your current directory that is called **cvY.txt**, where **Y** is the version number of the code-set-conversion object-file format. The **cvY.txt** file lists the code-set conversions in alphabetical order, sorted on the name of the object code-set-conversion file.

For two-way code-set conversions, the **\$INFORMIXDIR/gls/cvY** directory contains two code-set-conversion files. One file supports conversion from the characters in code set A to their mappings in code set B, and another supports the conversion in the return direction (from code set B to code set A). For more information on two-way code-set conversion, see [page A-14](#).

Figure A-4 shows a sample file, **cv9.txt**, that contains the available code-set conversions.

```

Filenames: cv9/002501b5.cvo and cv9/01b50025.cvo
Between Code Set: Greek
    and Code Set: IBM CCSID 437

Filenames: cv9/00250333.cvo and cv9/03330025.cvo
Between Code Set: Greek
    and Code Set: ISO8859-1
:
:
Filenames: cv9/033304e4.cvo and cv9/004e40333.cvo
Between Code Set: 8859-1
    and Code Set: 1252
:
:

```

Figure A-4
Sample glfiles File
for Informix
Code-Set-
Conversion Files

Examine the **cvY.txt** file to determine the code-set conversions that the **\$INFORMIXDIR/gls/cvY** directory on your system supports.

Listing Code-Set Files

When you specify the **-cm** command-line option, the **glfiles** utility creates a file that lists the available code-set files. For each **cmZ** subdirectory in **\$INFORMIXDIR/gls**, **glfiles** creates a file in the current directory that is called **cmZ.txt**, where **Z** is the version number of the code-set object-file format. The **cmZ.txt** file lists the code sets in alphabetical order, sorted on the name of the GLS object code-set file.

Figure A-5 shows a sample file, **cm3.txt**, that contains the available code sets.

```
Filename: cm3/032d.cm
Code Set: 8859-7

Filename: cm3/0333.cm
Code Set: 8859-1

Filename: cm3/0352.cm
Code Set: PC-Latin-1
:
:
Filename: cm3/04e4.cm
Code Set: CP1252
:
:
```

Figure A-5
*Sample glfiles File
for Informix
Code-Set Files*

Examine the **cmZ.txt** file to determine the code sets that the **\$INFORMIXDIR/gls/cmZ** directory on your system supports. ♦

Glossary

7-bit character	A character that is composed of seven bits, such as all the characters of the ASCII code set.
8-bit character	The 8-bit characters are single-byte characters with code values between 128 and 255. Examples from the ISO8859-1 code set include the non-English é, ñ, and ö characters. They can be interpreted correctly only if the software that interprets them is 8-bit clean.
8-bit clean	This term describes whether a piece of software or a file system can process character data that contains 8-bit characters. Otherwise, only the characters of the ASCII code set can be represented and processed correctly in such a system.
16-bit code set	In a 16-bit code set, (such as JIS X0208), approximately 65,000 distinct characters can be encoded.
alpha class	The alpha class of a code set consists of all characters that are classified as alphabetic. For example, the alpha class of the ASCII code set is the letters <i>a</i> through <i>z</i> and <i>A</i> through <i>Z</i> .
ALS	An acronym for Asian Language System. ALS refers to a class of products that have been developed to operate with multibyte code sets. ALS products support various multibyte code sets whose characters are composed of 8, 16, 24, and 32 bits. ALS servers and tools are available for the 6.x and earlier family of products. These products might have been developed with the GLS Library or other software written specifically to handle Asian language processing.

ASCII	An acronym for the American Standards Committee for Information Interchange. This acronym is often used to describe an ordered set of printable and nonprintable characters used in computers and telecommunication. This code set is traditionally used in computer systems found in the United States. It contains 128 characters (each of which can be represented with 7 bits of information) and is the proper subset of every GLS character map and logical unit.
byte	The smallest computer memory unit (often called an octet).
character	A logical unit of storage for the value code in a code set. It can be represented by one or more bytes and can be a numeric, alphabetic, or nonprintable character (control character).
client	An application program that requests services from a server program, typically a file server or a database server.
client locale	The environment that defines the behavior of the client application by specifying a language, a code set, and the conventions used for a particular language. These conventions can include date, time, and monetary formats. For example, to read and write files, the client refers to the client locale, which is typically specified by the <code>CLIENT_LOCALE</code> environment variable.
code point	An entry in a code set. For example, in the ASCII code set, the 65th code point is A.
coded character set	See code set.
code set	A given language has a character set of one or more natural-language alphabets plus additional symbols for digits, punctuation, and diacritical marks. Each character has at least one code set, which maps its characters to unique bit patterns. ASCII, ISO8859-1, Microsoft 1252, and EBCDIC are examples of code sets for the English language.
code-set conversion	A translation of code points from one code set to another, such as ASCII to EBCDIC.
code-set order	The sequence of characters when sorted in terms of their numerical representation and code-point value in a code set. For example, in the ASCII code set, uppercase characters (A through Z) are ordered before lowercase characters (a through z).

collation	The process of ordering or comparing characters. The order or compare process is based on the code-set value of the character (code-set order) or on the localized order (collated order). Full internationalization of sorting and collating functions and support libraries is required in order to correctly and efficiently perform case conversions, comparisons, and regular expressions such as <code>column1 = column2</code> .
collation order	The sequence of values that specifies some logical ordering in which the character fields in a database are sorted and indexed. Collation order is also known as collating sequence.
DB_LOCALE	On the client, the value of <code>DB_LOCALE</code> specifies how databases are created on the database server (that is, which code set and collation order). In addition, <code>DB_LOCALE</code> specifies the code-set conversion (if any) between the client application and database server. On the database server, <code>DB_LOCALE</code> specifies how databases are created if this information is not specified by the client application.
encoding	The process of mapping a national character to its numeric representation in a coded-character set.
fixed-width character	Fixed-width character implementations (for example, UNICODE) assign a fixed number of bytes for each character encoded. Fixed-width character implementations are also referred to as wide-character encoding, which typically implies the use of specially defined data types.
flexible-width character	Flexible-width character implementations support encoding schemes that allow for single-byte characters and multibyte characters to be used simultaneously. The simultaneous use of single-byte and multibyte characters is achieved by establishing a protocol for recognizing the respective code sets supported in the encoded scheme. An example of flexible-width character implementation is Extended UNIX Code (EUC). EUC allows for support of up to four code sets simultaneously. In EUC, the 7-bit ASCII code set is supported with up to three other code sets for interoperability and compatibility. The other three code sets can be encoded as two-, three-, or four-byte code sets.
Global Language Support	Global Language Support (GLS) is an application environment that allows Informix application-programming interfaces (APIs) and database servers to handle different languages, cultural conventions, and code sets. Products labeled GLS use the GLS Library and GLS locales to facilitate locale-based processing. The GLS library supports text processing and multibyte characters.
GLS	See <i>Global Language Support</i> .

Internationalization (I18n)	Internationalization (I18n) is the process of making Informix products easily adaptable to any culture and language. Among other features, internationalized software provides support for culturally specific sorting and for adaptable date, time, and money formats. An internationalized product does not require re-compilation of source code for each cultural environment. The user simply exchanges resource files and sets up the proper operating environment.
ISO8859-1	A code set that contains 256 single-byte characters. Characters 0 through 127 are the ASCII characters. Characters 128 through 255 are mostly characters from European languages, for example, é, ñ and ö.
language supplement	The result of the product localization process. A language supplement for a specific Western European language can be installed with an Informix product to allow the user to see error and warning messages in a language other than English. If installed with DB-Access, the menu names, options and on-line help for that product also appear in the specific language.
local character	A character in the native-language character set.
locale	The environment that defines the behavior of the program at runtime. The rules are usually based on the linguistic customs of the region or the territory. GLS library-based products use GLS locales that Informix defines and preprocesses for customers. Customers are not able to modify Informix locales. The locale can be set through an environment variable that dictates output formats for numbers, currency symbols, dates, and time as well as collation order for character strings and regular expressions.
localization (L10n)	Localization (abbreviated L10n) is the process adapting an internationalized product to a specific cultural environment. This process usually involves the creation of culturally specific resource files; the selection of message catalogs; the setting of date, time, and money formats; and the translation of the product user interface. It might also include the translation and production of end-user documentation, packaging, and collateral materials.
localization kit	The files and documentation that Informix supplies a localization center to enable translation to localize for the following products: C-ISAM, ESQL/C, ESQL/COBOL, OnLine, SE, and Universal Server.

multibyte character	<p>If a language contains more than 256 characters, the code set must contain multibyte characters. A multibyte character might require from two to four bytes of storage. Many Asian languages contain 3,000 to 8,000 ideographic characters. Such languages have code sets made up of both single byte and multibyte characters (a multibyte code set). Some characters in the Japanese SJIS code set are multibyte characters of two or three bytes. Applications that handle data in a multibyte code set cannot assume that one character requires only one byte of storage.</p>
native language	<p>The computer user's spoken or written language (for example, English, Chinese, German, and so on).</p>
NLS	<p>An acronym for Native Language System (or Native Language Support). Informix limits its use of this term to the support of English and European languages for Informix servers 6.x and higher.</p>
NLS Open, Implicit, Explicit	<p>The Informix 6.0 server and connectivity products introduced the NLS feature. The NLS feature was introduced in such a way that all types of client applications could access NLS data from the database server regardless of whether the client itself was able to process the NLS data according to the rules of a locale or whether the client supported the new NCHAR and NVARCHAR data types introduced by the database server.</p> <p>To process these new data types, the Informix 6.0 server and connectivity products created three modes by which a client could connect to a database: Open NLS, Implicit NLS and Explicit NLS. These modes are determined by the values of two environment variables (DBNLS and COLLCHAR) sent to the server.</p>
OPEN NLS	<p>This mode is for tools that have <i>not</i> been modified to process NLS data according to the rules of a locale and have <i>not</i> been modified to support the NCHAR and NVARCHAR data types.</p> <p>In this mode, the database server allows the client to connect to any database, regardless of the locale of the database. When NCHAR and NVARCHAR data is sent from the database server to the client application, the database server converts it to CHAR and VARCHAR data, respectively. When CHAR and VARCHAR data is sent from the client to the database server, the database server converts it to NCHAR and NVARCHAR data, respectively. This mode is enabled when the client sets DBNLS=2 and COLLCHAR=1.</p>

IMPLICIT NLS This mode is for tools that have been modified to process NLS data according to the rules of a locale but have *not* been modified to support the NCHAR and NVARCHAR data types.

In this mode, the locale used by the client must equal the locale used by the database to which the client is trying to connect. When NCHAR and NVARCHAR data is sent from the database server to the client application, the server converts it to CHAR and VARCHAR data, respectively. When CHAR and VARCHAR data is sent from the client to the database server, the database server converts it to NCHAR and NVARCHAR data, respectively. This mode is enabled when the client sets `DBNLS=1` and `COLLCHAR=1`.

EXPLICIT NLS This mode is for tools that have been modified to process NLS data according to the rules of a locale and have been modified to support the NCHAR and NVARCHAR data types.

In this mode, the locale that the client uses must equal the locale used by the database to which the client is trying to connect. This mode is enabled when the client sets `DBNLS=1` and `COLLCHAR=0`.

The following environment variable settings are correct, but they do not allow access to any NLS features: `DBNLS=0` and `COLLCHAR=0`. If you do not set these variables, the default is 0.

The following environment variable settings are not valid: `DBNLS=0` and `COLLCHAR=1` and `DBNLS=2` and `COLLCHAR=0`.

partial character A multibyte character that has lost one or more bytes so that the intended meaning of the character is lost. GLS software provides context-specific solutions that prevent partial characters from being generated during string-processing operations.

server locale The locale with which the server performs input and output. For example, files that are written and read with respect to the locale that the **SERVER_LOCALE** environment variable specifies.

server processing locale The environment that the database server dynamically determines based on the client locales and information that is stored in the database being accessed.

single-byte character	The number of unique characters in the language determines the amount of storage that each code-set character requires. Because a single byte can store values in the range of 0 to 255, it can uniquely identify 256 characters. Most Western languages have fewer than 256 characters, so their code sets consist of single-byte characters. When an application handles data such in these code sets, it can assume that one character is always stored in one byte.
UNICODE	A consortium of computer vendors defined UNICODE to create a unified worldwide code set. UNICODE was integrated as part of the ISO international standard 10646-1 and is emerging as the code-set choice for Microsoft operating systems. Informix supports UNICODE and continues to support the diverse prevailing code sets for compatibility reasons and for the advanced processing requirements for three-byte and four-byte encoding schemes.
white space	White space is a series of one or more space characters. The GLS locale defines the characters that are considered to be space characters. For example, both the <code>TAB</code> and blank might be defined as space characters in one locale, but certain combinations of the <code>CTRL</code> key and another character might be defined as space characters in a different locale file.
wide character	A wide-character form of a code set involves normalizing the size of each multibyte character so that each character is the same size. This size must be equal to or greater than the largest character that an operating system can support, and it must match the size of an integer data type that the C compiler can scale. Some examples of an integer data type that the C compiler can scale are short integer (short int), integer (int), or long integer (long int).

Index

A

ALS. *See* Asian Language Support (ALS).
 ALTER TABLE statement 3-11, 3-48
 Alternative date formats 2-29
 Alternative time formats 2-37
 ANSI compliance
 level Intro-18
 as formatting character 7-20
 ASCII code set 1-19
 Asian date. *See* Era-based dates.
 Asian Language Support (ALS)
 code-set conversion with 1-46
 collation order 1-12
 DBDATE support 2-7, 7-12
 DBTIME support 2-20
 information in systables 1-26
 multibyte data 1-3
 precedence of environment
 variables 1-25, 1-38, 1-40
 reverting to 5-10
 Asian language. *See* Multibyte character.

B

BETWEEN conditions 3-25
 Binary large object (blob) 1-48, 4-7, 8-13
 See also BYTE data type, TEXT data type.
 blob. *See* Binary large object (blob).

BYTE data type
 code-set conversion 1-48, 4-7, 8-12
 partial characters 3-39
 See also Binary large object (blob).

C

.c file extension 7-4, 8-11
 C compiler
 8-bit clean 8-8
 limitations 7-4, 8-7
 multibyte characters 8-7, 8-8
 non-ASCII filenames 7-4
 non-ASCII source code 8-7, 8-8
 See also ESQL/C compiler.
 Cast 4-17
 CC8BITLEVEL environment
 variable 2-5, 7-4, 8-10
 CCSID code set. *See* IBM CCSID code set.
 CHAR data type
 code-set conversion 1-48, 4-7
 collation order 1-12
 difference from NCHAR 3-12
 GLS aspects 3-17
 performance considerations 3-16
 Character
 7-bit 1-8
 8-bit 1-9
 mismatched 1-42, A-17
 multibyte. *See* Multibyte character.
 non-ASCII. *See* Non-ASCII character.
 nonprintable 3-13, 3-15

- partial 3-36, 7-25
- shifting case of 7-23
- single-byte. *See* Single-byte character.
- white space. *See* White space.
- Character classification. *See* Locale, CTYPE category.
- Character data
 - avoiding corruption of 1-48, 4-7
 - collation of 1-36, 3-20, A-6
 - converting 1-41, 1-48, 4-7
 - data types 3-11
 - equivalent characters 1-11, 3-23, 3-28, A-6
 - ESQL functions 7-23
 - interpreting 1-26, 1-36
 - processing with locales 1-5
 - See also* Data.
- Character set 1-8, A-18
- Charmap file. *See* Code-set file.
- CHAR_LENGTH function 3-45
- Chinese locale 1-7, 1-30
- chkenv utility 4-11, 5-10
- Chunk 4-4, 5-4
- Client application
 - checking a connection 1-33, 1-38, 1-47
 - code-set conversion 1-45, 1-46
 - definition of 1-6
 - end-user formats 1-12
 - establishing a connection 7-24
 - opening another database 1-38, 1-47
 - requesting a connection 1-31, 2-18
 - sending client locale to
 - server 1-32, 1-39, 7-24
 - setting a locale 1-18, 1-24, 1-28
 - support for locales 1-6, 1-7
 - uses of client locale 1-23, 2-6
 - uses of database locale 1-27
 - verifying locales 1-45
 - See also* ESQL/C program; ESQL/COBOL program.
- Client code set 1-41, 1-45, 1-46
- Client computer
 - client code set 1-41
 - code-set-conversion files 1-46
 - setting CLIENT_LOCALE 1-28
 - setting DB_LOCALE 1-28
- Client locale
 - client code set 1-41
 - code set. *See* Client code set.
 - COLLATION category 1-24
 - CTYPE category 1-24
 - customizing 1-49
 - definition of 1-23, 1-26
 - determining 1-24
 - ESQL/C source files 7-5, 7-6
 - ESQL/COBOL source files 7-5, 7-6
 - MESSAGES category A-9
 - MONETARY category 1-24
 - NUMERIC category 1-24
 - sample 1-23, 1-24, 1-35
 - sending to database server 1-32, 7-24
 - setting 1-28
 - TIME category 1-24
 - See also* Client application; CLIENT_LOCALE environment variable.
- Client/server environment
 - client locale 1-32
 - code-set conversion 1-41, 1-44
 - database locale 1-33
 - locales of 1-18, 1-22
 - server-processing locale 1-34
 - setting environment
 - variables 1-28
- CLIENT_LOCALE environment
 - variable
 - client code set 1-41
 - default value 1-28
 - ESQL filenames 7-4
 - ESQL source code 7-4
 - for NLS applications 1-37
 - interpreting command-line
 - arguments 4-10, 5-9, 6-9
 - location of message files 2-16
 - precedence of 1-24, 1-39, 1-51, 1-52, 1-53, 2-16, 7-14, 7-17, 7-19
 - role in code-set conversion 1-45, 1-47
 - sending to database server 1-32
 - setting 1-28
 - syntax 2-6
- with TEXT data 3-12, 3-15, 3-17, 3-18, 3-19
- See also* Client locale.
- .cm file extension A-18, A-21
- .cmo file extension A-18
- cmZ.txt file A-26
- .cob file extension 7-4
- COBOL compiler 7-4
- Code point 1-8, 1-10, 3-16
- Code set
 - 1252 1-8, 1-19, 1-21
 - 8859-1 1-8, 1-19, 1-21, A-13
 - affecting filenames 2-25, 4-6, 5-6, 6-5, 7-5
 - ASCII 1-8, 1-19
 - code points 1-8, 3-16
 - compatible 1-6, 1-26
 - condensed name 1-21, 1-26, 1-49, A-12
 - convertible 1-29, 1-46, 1-48, 4-7
 - default 1-19, 1-20, 1-30
 - definition of 1-8
 - determining 1-30, 1-41
 - for client application 1-23
 - for database 1-26
 - for database server 1-27
 - in locale name 1-20, 1-21, 1-33
 - incompatible 1-46
 - multibyte 1-9, 1-30, 3-35, 3-36, 3-42, 7-25
 - single-byte 1-8, 1-30, 3-35, 3-38, 3-41
 - source 1-41, 1-42
 - target 1-41, 1-42
 - See also* Client code set; Code-set conversion; Database code set; Server code set.
- Coded Character Set Identifier (CCSID). *See* IBM CCSID code set.
- Code-set conversion
 - by client application 1-45
 - by database server 1-47
 - character mismatches 1-42, A-17
 - data converted 1-48, 4-7
 - definition of 1-41
 - files. *See* Code-set-conversion file.
 - for blob data 1-48, 4-7, 8-13
 - for column names 1-48, 4-8

- for cursor names 1-48, 4-8
- for error message text 1-48, 4-8
- for LVARCHAR 4-7
- for SQL data types 1-48, 4-7
- for SQL statements 1-48, 4-8
- for statement IDs 1-48, 4-8
- for stored procedure text 1-48, 4-8
- for table names 1-48, 4-8
- handling mismatched characters 1-42
- in ESQL/C program 8-12
- limitations 1-41
- lossy error 1-42
- performing 1-43, 1-47, 1-48, 4-7
- registry file A-18, A-20
- role of CLIENT_LOCALE 1-45, 1-47
- role of DB_LOCALE 1-45, 1-47
- role of SERVER_LOCALE 1-47
- two-way A-16
- Code-set file
 - description of A-17
 - listing A-26
 - location of A-18, A-19
 - object A-18
 - removing A-21
 - source A-18
- Code-set order. *See* Collation order.
- Code-set-conversion file
 - description of A-14
 - listing 1-46, A-25
 - location of A-15
 - object A-15, A-20
 - removing unused A-20
 - source 1-42, A-15, A-20
- Collation
 - definition of 1-10
 - equivalence classes 1-11, 3-23, 3-28, 3-31, A-6
 - of character data 3-20
 - of NCHAR 3-12
 - of NVARCHAR 3-14
 - sort order. *See* Collation order.
 - with ALS database server 1-12, 1-26
 - with NLS database server 1-12, 1-26

- COLLATION locale category
 - description of A-4, A-6
 - in client locale 1-24
 - in default locale A-4
 - in locale source file A-12
 - in server-processing locale 1-38
- Collation order
 - code-set 1-10, 1-12, 3-16
 - localized 1-5, 1-11, 1-12, 1-20, 1-36, 3-16
 - tasks affected by 1-10
 - types of 1-10
- Column (database)
 - expressions 3-34
 - in code-set conversion 1-48, 4-8
 - naming 1-5, 1-6, 1-7, 3-5, 7-6
 - substrings 3-34, 3-39
- Command-line arguments 4-10, 5-9, 6-9
- Command-line conventions
 - elements of Intro-9
 - example diagram Intro-10
 - how to read Intro-10
- Comment icons Intro-7
- Comments 2-5, 3-34, 7-6
- Compiler. *See* C compiler; ESQL/C compiler.
- Compliance, with industry standards Intro-18
- Conditions
 - BETWEEN 3-25
 - IN 3-27
 - LIKE 3-30
 - MATCHES 3-28
 - relational operator 3-24
- CONNECT statement 3-5
- Connection. *See* Database server connection.
- Constraint 3-5, 7-6
- Conversion modifier 1-51, 2-29, 2-37
- CREATE CAST statement 4-17
- CREATE DATABASE statement 3-6
- CREATE DISTINCT TYPE statement 4-17

- CREATE FUNCTION statement 4-17
- CREATE INDEX statement 3-4, 3-6, 3-20
- CREATE OPAQUE TYPE statement 4-17
- CREATE OPCLASS statement 4-18
- CREATE PROCEDURE statement 3-6, 3-7, 4-18
- CREATE ROLE statement 3-6
- CREATE ROW TYPE statement 4-18
- CREATE SYNONYM statement 3-7
- CREATE TABLE statement
 - character data types 3-11
 - column name in 3-5
 - constraint name in 3-5
 - MONEY columns 3-48
 - naming database objects 3-4
 - table name in 3-7
- CREATE TRIGGER statement 3-7
- CREATE VIEW statement 3-7
- CTYPE locale category
 - character case 7-23
 - description of A-4, A-6
 - in client locale 1-24
 - in default locale A-4
 - in locale source file A-12
 - in server-processing locale 1-38
 - white-space characters 2-26, 2-35
- Currency data. *See* Monetary data.
- Currency notation 1-15, 1-53, 2-19
- Currency symbol 1-15, 1-20, 3-49, 7-19, A-7
- Cursor 1-5, 1-6, 1-7, 1-48, 3-5, 4-8, 7-6
- .cv file extension 1-42, A-15, A-20
- .cvo file extension A-15, A-20
- cvY.txt file A-25
- .c_ file extension 8-11

D

.dat file extension 6-3

Data

- character 3-11

- converting 1-48, 4-7

- corruption 2-6, 2-18

- portability 6-6

- transferring 1-34

- See also* Character data; Date data;

- Monetary data; Numeric data;

- Time data.

Data type

- BLOB 4-13

- BYTE 1-48, 4-7

- CHAR 1-48, 3-17, 4-7

- character 3-11

- CLOB 4-13

- code-set conversion 1-48, 4-7

- collation order 1-12

- complex 4-14

- DATE 1-15, A-9

- DATETIME 1-15, A-9

- DECIMAL 1-14, A-7

- distinct 4-14

- FLOAT 1-14, A-7

- INTEGER 1-14, A-7

- internal format 1-12

- locale-sensitive 1-26, 3-11, 8-6

- locale-specific 1-36

- locator structure 8-12

- LVARCHAR 4-13

- MONEY 1-14

- NCHAR 1-48, 3-11, 4-7, 8-6

- numeric 1-14, A-7

- NVARCHAR 3-13, 8-6

- NVARCHAR data type 1-48, 4-7

- opaque 4-14

- SMALLFLOAT 1-14, A-7

- SMALLINT 1-14, A-7

- smart large objects 4-13

- TEXT 1-48, 3-19, 4-7

- VARCHAR 3-18

- VARCHAR data type 1-48, 4-7

- See also individual data type names.*

Database

- loading 3-52

- migrating from GLS 5-10

- naming 3-6, 6-4, 6-7, 7-6

- saving locale information 1-26

- SE names 6-4, 6-6

- unloading 3-52

- Database code set 1-41, 1-45, 1-46

- Database conversion 1-26, 5-10

- Database cursor. *See* Cursor.

Database locale

- code set. *See* Database code set.

- database code set 1-41

- in system catalog 1-26, 1-33

- incompatible 1-33

- sample 1-23, 1-27, 1-35

- saving 1-26

- setting 1-28

- uses of 1-36

- verifying 1-33, 1-38, 2-18

- See also* DB_LOCALE
environment variable.

- Database objects 1-6, 3-4

Database server

- code-set conversion 1-47

- collation 1-12

- determining server-processing

- locale 1-31, 1-34

- end-user formats 1-13

- internal formats 1-12

- interpreting character data 1-26

- setting a locale 1-28

- setting locale 1-18

- support for locales 1-5, 1-7

- uses of client locale 1-24, 1-31, 1-32

- uses of database locale 1-26

- uses of server locale 1-27, 4-3, 5-3,
6-3

- using DB_LOCALE 2-18

- utilities 1-6

- verifying a connection 1-31, 7-24

- verifying database locale 1-33,
1-38

- See also* INFORMIX-OnLine

- database server;

- INFORMIX-SE database

- server; INFORMIX-Universal

- Server.

Database server connection

- client-locale information 1-32

- establishing 1-31, 7-24

- naming 3-5

- sample 1-23, 1-24, 1-27, 1-44

- server-processing locale 2-6

- verifying 1-31, 1-33, 1-38, 7-24

- warnings 1-33, 7-24

Date data

- alternative formats 2-29

- Asian. *See* Era-based dates.

- customizing format of 1-50

- end-user format 1-15, 1-19, 1-39,
1-50, A-9

- format of A-8

- locale-specific 1-6, 1-14

- precedence of environment

- variables 1-51, 7-14

- setting GL_DATE 2-25

- See also* Data; DATE data type;

- DATETIME data type; Era-
based dates.

DATE data type

- end-user format 1-15, 1-19, 1-50,
2-7, 2-25, A-9

- era-based dates 1-51

- ESQL library functions 7-8

- ESQL/C functions 7-8, 7-10

- ESQL/C library functions 7-9

- ESQL/COBOL library

- routines 7-9

- ESQL/COBOL routines 7-8, 7-10

- extended-format strings 7-10

- internal format 1-12, 1-15

- precedence of environment

- variables 1-51, 7-14

- See also* Date data.

DATETIME data type

- end-user format 1-15, 1-19, 1-50,
2-20, 2-35, A-9

- era-based dates 1-51

- ESQL library functions 7-15

- ESQL/C functions 7-15

- ESQL/COBOL routines 7-15

- extended-format strings 7-16

- formatting directives for 2-36

- internal format 1-15

- precedence of environment variables 1-52, 7-17
- See also* Date data.
- DB code set. *See* Database code set.
- DB-Access utility 1-6
- dbaccess utility 4-10, 5-10, 6-9
- DBAPICODE environment variable 1-45
- DBCENTURY environment variable 2-29
- DBCODESET environment variable 1-32, 1-37, 1-39
- DBDATE environment variable
 - era-based dates 1-51, 2-8, 3-51
 - ESQL library functions 7-9
 - precedence of 1-24, 1-39, 1-51, 7-14
 - sending to database server 1-32
 - setting 1-50
 - syntax 2-7
- dbexport utility 1-7, 2-25, 4-11, 5-10, 6-9
- dbimport utility 4-11, 5-10, 6-9
- DBLANG environment variable
 - precedence of 2-16
 - setting 1-49
 - syntax 2-15
- dbload utility 4-11, 5-10, 6-9
- DBMONEY environment variable
 - defining currency symbols 7-22
 - ESQL library functions 7-19, 7-22
 - precedence of 1-24, 1-39, 1-53, 7-19
 - sending to database server 1-32
 - setting 1-53
 - syntax 2-19
- DBNLS environment variable 1-24, 1-32, 1-34, 1-39, 1-45, 1-51, 1-52, 1-53
- .dbs file extension 6-3
- dbschema utility 4-11, 5-10, 6-9
- DBTIME environment variable
 - era-based dates 3-51
 - ESQL library functions 7-16
 - precedence of 1-24, 1-39, 1-52, 7-17
 - sending to database server 1-32
 - setting 1-50
 - syntax 2-20
 - with multibyte characters 2-22

- DB_LOCALE environment variable
 - database code set 1-41
 - default value 1-28
 - precedence of 1-37
 - role in code-set conversion 1-45, 1-47
 - sending to database server 1-32
 - server-processing locale 1-37
 - setting 1-28
 - syntax 2-17
 - verifying database locale 1-33
- See also* Database locale.
- DECIMAL data type 1-14, 1-53, A-7
- Decimal separator 1-14, 1-19, 3-49, 7-19, A-7
- DECLARE statement 3-5
- Default locale
 - default code set 1-19, 1-20, 1-30, A-13
 - definition of 1-18
 - for client application 1-28
 - for database server 1-28
 - locale modifier 1-21
 - locale name 1-20
 - required A-20
- DELETE statement
 - era-based dates 3-50
 - GLS considerations 3-49
 - WHERE clause conditions 3-50
- DELIMIDENT environment variable 3-10
- Delimited identifiers 3-10
- DESCRIBE statement 8-14
- Diagnostic file 1-27, 4-3, 5-3
- Distinct data type 4-17
- Documentation
 - related Intro-17
- Documentation conventions
 - command-line Intro-8
 - icon Intro-7
 - sample-code Intro-11
 - typographical Intro-6
- Documentation notes Intro-17
- Documentation, types of
 - documentation notes Intro-17
 - error message files Intro-16
 - machine notes Intro-17

- on-line manuals Intro-15
- printed manuals Intro-16
- related reading Intro-17
- release notes Intro-17
- Dollar (\$) sign
 - as formatting character 7-20
- dtevmasc() library function 7-15
- dttofmtasc() library function 7-15

E

- .ec file extension 7-4, 8-11
- .eco file extension 7-4
- ECO-DAT library routine 7-8, 7-9
- ECO-DEF library routine 7-8, 7-10
- ECO-DSH library routine 7-23
- ECO-DTCVASC library routine 7-15
- ECO-DTTOASC library routine 7-15
- ECO-FEL library routine 7-17
- ECO-FIN library routine 7-17
- ECO-FMT library routine 7-8, 7-10
- ECO-STR library routine 7-8, 7-9
- ECO-USH library routine 7-23
- End-user format
 - conversion modifier 2-29, 2-37
 - customizing 1-49
 - date data 1-15, 1-19, 1-50, 2-21, 2-26, 2-35, A-9
 - date format qualifiers 2-31
 - default 1-19
 - definition of 1-12, 1-50, 1-53
 - extended DATE-format strings 7-10
 - extended DATETIME format strings 7-16
 - monetary data 1-14, 1-19, 1-53, 2-19, A-8
 - numeric data 1-14, 1-19, A-7
 - printing 1-13, 1-14, 1-15, 2-33, 2-38
 - scanning 1-13, 1-14, 1-15, 2-32, 2-38
 - sending to database server 1-32, 1-39
 - time data 1-15, 1-19, 1-50, 2-35, A-9
 - time format qualifiers 2-38

- English locale 1-30, A-10
See also Default locale.
- Environment variable
 - CC8BITLEVEL 2-5
 - CLIENT_LOCALE 1-28, 2-6
 - DBAPICODE 1-45
 - DBCENTURY 2-29
 - DBCODASET 1-37, 1-39
 - DBDATE 2-7
 - DBLANG 2-15
 - DBMONEY 2-19
 - DBNLS 1-32
 - DBTIME 2-20
 - DB_LOCALE 1-28, 2-17
 - DELIMIDENT 3-10
 - ESQLMF 2-22
 - GLS8BITFSYS 2-23
 - GLS-related 2-4
 - GL_DATE 2-25
 - GL_DATETIME 2-35
 - LANG 1-32
 - locale-related 1-28
 - precedence for ALS 1-25, 1-38, 1-40
 - precedence for client locale 1-24
 - precedence for DATE data 1-51, 7-14
 - precedence for DATETIME data 1-51, 7-17
 - precedence for monetary data 1-53, 7-19
 - precedence for NLS 1-25, 1-37, 1-40
 - precedence for server-processing locale 1-37, 1-39
 - SERVER_LOCALE 1-28, 2-41
See also individual environment variable names.
- Era-based dates
 - DATE-format functions 7-8
 - DATETIME-format functions 7-15
- DBDATE formats 2-8, 7-9
- DBTIME formats 2-21, 7-16
- defined in locale A-8
- definition of 1-16
- extended-format strings 7-10, 7-16
- GL_DATE formats 1-51, 2-29
- GL_DATETIME formats 1-51
 - in DELETE statement 3-50
 - in INSERT statement 3-50
 - in SQL statements 3-50
 - in UPDATE statement 3-50
- sample 1-16, 2-10, 2-12, 2-13
- Error message
 - DATE-format 7-23
 - DATETIME-format 7-23
 - GLS-specific 7-23
 - in code-set conversion 1-48, 4-8
 - numeric-format 7-23
- Error message files Intro-16
- Escape character 3-32
- esql command. *See* ESQL/C processor.
- ESQL library functions
 - currency notation in 7-18, 7-20
 - DATE-format functions 7-8
 - DATETIME-format functions 7-15
 - GLS enhancements 7-7
 - GLS error messages 7-23
 - numeric-format functions 7-17
 - string functions 7-23
- ESQL program. *See* ESQL/C program; ESQL/COBOL program.
- esqlc command. *See* ESQL/C preprocessor.
- esqlcobol command. *See* ESQL/COBOL processor.
- ESQLMF environment variable 2-22, 8-10
- esqlmf filter. *See* ESQL/C filter.
- ESQL/C compiler
See also C compiler.
- ESQL/C data types 1-48, 4-7, 8-5
- ESQL/C filter
 - description of 8-7
 - invoking 8-9
 - non-ASCII characters 8-8
 - with CC8BITLEVEL 2-5, 8-10
 - with ESQLMF 2-22, 8-10
- ESQL/C function library
 - DATE-format functions 7-8, 7-9, 7-10
 - DATETIME-format functions 7-15
- dtecvfmtasc() 7-15
- dttofmtasc() 7-15
- era-based dates 7-9, 7-10
- precedence for DATE data 7-14
- precedence for DATETIME data 7-17
- precedence for MONEY data 7-19
- rdatestr() 7-8, 7-9
- rdefmtdate() 7-8, 7-10
- rdownshift() 7-23
- rfmtdate() 7-8, 7-10
- rfmtdec() 7-17
- rfmtdouble() 7-17
- rfmtlong() 7-17, 7-18, 7-20
- rstrdate() 7-8, 7-9
- rupshift() 7-23
- ESQL/C library functions
 - rdatestr() 1-13
- ESQL/C preprocessor 2-6, 8-7
- ESQL/C processor
 - definition of 7-3
 - invoking ESQL/C filter 2-5, 8-10
 - multibyte characters 2-25, 7-4
 - non-ASCII filenames 2-25, 7-4
 - non-ASCII source code 8-10
 - operating-system files 7-4
 - with CC8BITLEVEL 2-5
 - with ESQLMF 2-22, 8-10
- ESQL/C program
 - accessing NCHARs 8-6
 - accessing NVARCHARs 8-6
 - checking database connection 1-33, 7-25
 - comments 2-5, 7-6
 - compiling 8-9, 8-11
 - data type constants 8-13, 8-17, 8-18
 - filenames 7-7
 - handling code-set conversion 8-12
 - host variables 1-23, 7-6
 - indicator variables 7-6
 - literal strings 1-12, 1-23, 2-5, 7-7
 - non-ASCII source code 7-6
 - writing blobs to database 8-12
 - See also* Client application.
- ESQL/COBOL processor 2-25, 7-3, 7-4

ESQL/COBOL program
 checking database
 connection 1-33, 7-25
 comments 7-6
 filenames 7-7
 host variables 1-23, 7-6
 indicator variables 7-6
 literal strings 1-12, 1-23, 7-7
 non-ASCII source code 7-6
See also Client application.

ESQL/COBOL routine library
 DATE-format routines 7-8, 7-9,
 7-10
 DATETIME-format routines 7-15
 ECO-DAT 7-8, 7-9
 ECO-DEF 7-8, 7-10
 ECO-DSH 7-23
 ECO-DTCVASC 7-15
 ECO-DTTOASC 7-15
 ECO-FEL 7-17
 ECO-FIN 7-17
 ECO-FMT 7-8, 7-10
 ECO-STR 7-8, 7-9
 ECO-USH 7-23
 era-based dates 7-9, 7-10
 precedence for DATE data 7-14
 Explain file 1-27

F

Feature icons Intro-8
 FETCH statement 3-6
 File
 cmZ.txt A-26
 code-set-conversion. *See* Code-
 set-conversion file.
 code-set. *See* Code-set file.
 cvY.txt A-25
 diagnostic 1-27, 4-3, 5-3
 ESQL source 7-3
 Informix-proprietary 1-27
 lcX.txt A-22, A-24
 LOAD FROM 3-52
 locale object file A-11
 locale source file A-11
 locale. *See* Locale file.
 log 1-27, 4-3, 5-3, 6-3
 message 1-27, 1-44, 1-49, 2-15

name of. *See* Filename.
 registry A-18, A-20
 SE data 6-3
 SE index 6-3
 sqexplain.out 1-27
 text 3-51
 UNLOAD TO 3-52

File extension
 .c 7-4, 8-11
 .cm A-18, A-21
 .cmo A-18
 .cob 7-4
 .cv 1-42, A-15, A-20
 .cvo A-15, A-20
 .c_ 8-11
 .dat 6-3
 .dbs 6-3
 .ec 7-4, 8-11
 .eco 7-4
 .idx 6-3
 .iem 2-17
 .lc A-11, A-12, A-15, A-20
 .lco A-11, A-20
 .o 8-11

Filename
 7-bit clean 2-24
 8-bit 6-6
 8-bit clean 2-24, 4-5, 5-5, 6-4, 7-5
 generating 2-23, 4-4, 5-4, 6-4, 7-4
 illegal characters in 2-24
 multibyte. *See* Filename, non-
 ASCII.
 non-ASCII 2-23, 3-6, 4-4, 5-4, 6-4,
 7-4, 7-7
 validating 4-7, 5-7, 6-6

FLOAT data type 1-14, 1-53, A-7

Formatting directive
 conversion modifier 1-51
 conversion modifiers 2-30
 field precision 2-33, 2-39
 field specification 2-32, 2-33, 2-39
 field width 2-33, 2-39
 white space 2-28
 with DBTIME 2-21
 with GL_DATE 2-26
 with GL_DATETIME 2-35

Format. *See* End-user format.

French locale 1-13, 1-14, 1-21, 1-30,
 1-33, 1-38, 1-47, A-10

G

Gengo year format 1-17
 German locale 1-24, 1-27, 1-30, A-10
 glfiles utility
 -cm option A-22, A-26
 code-set files A-26
 code-set-conversion files 1-46,
 A-25
 -cv option A-22, A-25
 -lc option A-22
 locale files 1-21, A-22
 sample output A-23, A-24, A-25,
 A-26
 syntax A-21

Global Language Support. *See* GLS
 environment.

GLS Application Programming
 Interface 4-18

GLS environment
 locale. *See* Locale.

GLS feature
 available locales 1-21, 1-29
 CHAR data type 3-17
 character data types for host
 variables 8-6
 client/server environment 1-18,
 1-22
 description of 1-3
 environment variables 2-4
 ESQL library functions 7-7
 for ESQL/C SQL API 7-3, 8-3
 for ESQL/COBOL SQL API 7-3
 for INFORMIX-Universal
 Server 4-3
 for OnLine database server 5-3
 for SE database server 6-3
 for SQL 3-3
 functionality listed 1-5
 fundamentals 1-3
 GLS files A-9, A-15, A-18, A-19
 GLS library 1-4
 locales 1-8
 managing GLS files A-1
 NCHAR data type 3-11
 NVARCHAR data type 3-13
 TEXT data type 3-19
 using character data types 3-11
 VARCHAR data type 3-18

GLS locale. *See* Locale.
GLS8BITFSYS environment variable
 for ESQL filenames 7-5
 for INFORMIX-Universal Server filenames 4-5
 for OnLine filenames 5-5
 for SE filenames 6-4, 6-6
 syntax 2-23
GL_DATE environment variable
 era-based dates 1-51, 3-51
 ESQL library functions 7-8
 precedence of 1-24, 1-39, 1-51, 7-14
 sending to database server 1-32
 setting 1-50
 syntax 2-25
GL_DATETIME environment variable
 era-based dates 3-51
 era-based dates and times 1-51
 ESQL library functions 7-15
 precedence of 1-24, 1-39, 1-52, 7-17
 sending to database server 1-32
 setting 1-50
 syntax 2-35

H

Host variable
 end-user formats 1-12
 ESQL/C example 7-6, 8-4
 naming 1-7, 3-6, 7-6, 8-4

I

IBM CCSID code set
 437 1-43, A-16
 819 A-13, A-16, A-19
 definition of 1-43
Icons
 comment Intro-7
 feature Intro-8
 platform Intro-8
Identifier 1-31, 3-10
 See also Delimited identifiers; SQL identifier.
.idx file extension 6-3

.iem file extension 2-17
IN conditions 3-27
Index 3-6, 6-4
Indicator variable 1-7, 7-6, 8-5
Industry standards, compliance with Intro-18
Informix Code-Set Name-Mapping file. *See* registry file.
Informix code-set-conversion file. *See* Code-set-conversion file.
INFORMIXDIR environment variable
 location of code-set files A-18, A-26
 location of code-set-conversion files A-15, A-25
 location of locale files 1-21, A-9, A-22
 location of message files 2-15, 2-16
 location of operating-system locales A-11
 location of registry file A-19
 with glfiles A-22
INFORMIX-OnLine Dynamic Server
 checking filenames 5-7
 chunk name 5-4
 code-set conversion 1-44
 diagnostic files 5-3
 identifiers 1-31
 log filename 5-4
 message log file 5-3
 multibyte characters 5-8
 multibyte filenames 2-25, 5-4
 NCHAR sizing restrictions 3-12
 non-ASCII filenames 2-25, 5-4
 operating-system files 5-3
 pathnames 5-4
 sample connection 1-22
 support for locales 1-5
 uses of server locale 5-3
 using GLS8BITFSYS 2-25, 5-5
 utilities 5-8
 with TRIM function 8-17, 8-18
 See also Database server.

INFORMIX-SE database server
 compatibility issues 6-6
 data file 6-3
 data portability 6-6
 identifiers 1-31
 index file 6-3
 log file 6-3
 multibyte characters 6-6, 6-8
 multibyte filenames 2-25
 naming a database 6-7
 NCHAR sizing restrictions 3-12
 non-ASCII filenames 2-25, 6-4
 operating-system files 6-3
 support for locales 1-5
 uses of server locale 6-3
 using GLS8BITFSYS 2-25, 6-4
 utilities 6-8
 with TRIM function 8-17
 See also Database server.
INFORMIX-Universal Server
 checking filenames 4-7
 chunk name 4-4
 diagnostic files 4-3
 log filename 4-4
 message log file 4-3
 multibyte characters 4-8
 multibyte filenames 4-4
 non-ASCII filenames 4-4
 pathnames 4-4
 uses of server locale 4-3
 using GLS8BITFSYS 4-5
 utilities 4-8
INSERT statement
 embedded SELECT 3-50
 end-user formats 1-13
 era-based dates 3-50
 GLS considerations 3-49
 specifying quoted strings 3-33
 VALUES clause 3-50
INTEGER data type 1-14, A-7
ISO8859-1 code set 1-19, 1-21

J

Japanese Imperial dates 1-16, 1-17, 1-51, 2-12, 2-13
Japanese locale 1-7, 1-28, 1-29, 1-30, 1-34, 1-38, 1-47

K

Korean locale 1-7, 1-30

L

LANG environment variable
precedence of 1-25, 1-39, 1-52,
1-53, 2-16, 2-17
sending to database server 1-32

Language
code sets 1-43
default 1-20
for client application 1-23
for database 1-26
for database server 1-27
in locale name 1-20, 1-33, A-10

.lc file extension A-11, A-12, A-15,
A-20

.lco file extension A-11, A-20

lcX.txt file A-22, A-24

LC_COLLATE environment
variable 1-24

LC_COLLATION environment
variable 1-37

LC_CTYPE environment
variable 1-24, 1-37

LC_MONETARY environment
variable 1-25, 1-39, 1-53

LC_NUMERIC environment
variable 1-25, 1-39

LC_TIME environment
variable 1-25, 1-39, 1-51
precedence 1-52

LC_* environment variables. *See the*
LC_COLLATE, LC_CTYPE,
LC_MONETARY,
LC_NUMERIC, LC_TIME
environment variables
individually.

LENGTH function 3-40

LIKE relational operator 1-10, 3-30

Literal matches 3-28, 3-31

Literal string 1-12, 2-5, 7-7

Load file 3-52

LOAD statement 3-6, 3-49, 3-52

Locale
alpha class 3-8
code set. *See* Code set.
COLLATION category. *See*
COLLATION locale category.
CTYPE category. *See* CTYPE
locale category.
default. *See* Default locale.
definition of 1-4, 1-8, 1-18
environment variables 1-28
filename A-9, A-12
for code-set conversion 1-40
for database server
connections 1-31
identifying. *See* Locale name.
International Language
Supplement 1-7
listing 1-21, A-21
locale categories A-4
locale names 1-20
MESSAGES category. *See*
MESSAGES locale category.
MONETARY category. *See*
MONETARY locale category.
name. *See* Locale name.
non-ASCII characters 1-30
NUMERIC category. *See*
NUMERIC locale category.
operating-system A-24
operating-system
compatible A-11
sample 1-30
setting 1-18, 1-28
TIME category. *See* TIME locale
category.
uses of 1-30
U.S. English. *See* Default locale.
verifying 1-33, 1-38
white space Intro-14
See also Client locale; Database
locale; Server locale.

Locale category. *See* Locale, locale
categories.

Locale file
description of 1-21, A-3
listing 1-21, A-21, A-22
location of 1-21, A-9

object A-11, A-20

operating-system
compatible A-11
removing unused A-20
required A-20
source A-11, A-20

Locale modifier 1-20, 1-21, 1-33,
A-12

Locale name
assigning 1-28
code-set name 1-20, 1-21, 1-30,
1-33
description of 1-20
language name 1-20, 1-33, A-10
locale modifier name 1-20, 1-21,
1-33, A-12
sample 1-21
syntax 1-20
territory name 1-20, 1-33, A-10

Localized collation order. *See*
Collation order, localized.

Locator structure 8-13

loc_buffer field 8-14

loc_t data type 8-12, 8-13

loc_type field 8-13

Log file 1-27, 4-3, 5-3, 6-3

Log filename, non-ASCII characters
in 4-4, 5-4

Lossy error 1-42

LVARCHAR data type
code-set conversion 4-7

M

Machine notes Intro-17

MATCHES relational operator 1-10,
3-28

Message file
compiled 2-17
error messages Intro-16
language-specific 2-15
localized 1-49
locating at runtime 2-16
sample 1-27
specifying location of 1-49, 2-15

Message log
sample 1-44

MESSAGES locale category
 description of A-4, A-9
 in default locale A-5
 in locale source file A-12
 in server-processing locale 1-40

Microsoft 1252 code set 1-19, 1-21

Migrating
 from GLS databases 5-10
 to or from GLS databases 4-11, 5-10, 6-8, 6-9

Ming Guo year format 1-16, 1-51, 2-12

Modifier. *See* Locale modifier.

Monetary data
 currency notation 1-14, 3-49, A-7
 currency symbol 1-15, 1-20, 3-49, 7-19, A-7
 decimal separator 1-14, 1-19, 3-49, 7-19, A-7
 default scale 3-48
 end-user format 1-14, 1-19, 1-39, 1-53, A-8
 format of A-7
 locale-specific 1-6
 negative 1-15, 1-19, A-7
 positive 1-15, 1-19, A-7
 precedence of environment variables 1-53, 7-19
 thousands separator 1-15, 1-19, 3-49, 7-19, A-7
See also Data; MONEY data type.

MONETARY locale category
 currency symbol 7-20
 description of A-4, A-7
 end-user formats 1-15, A-7
 in client locale 1-24
 in default locale A-5
 in locale source file A-12
 in server-processing locale 1-40
 numeric-formatting functions 7-19

MONEY data type
 defining 3-48
 end-user format 1-14, 2-19
 internal format 1-14, 1-53, 3-48
 precedence of environment variables 1-53, 7-19
See also Monetary data.

Multibyte character
 column substrings 3-35
 definition of 1-9
 filtering 8-8
 in cast names 4-17
 in column names 1-5, 1-6, 1-7, 3-5, 7-6
 in comments 2-5, 7-6
 in connection names 3-5
 in constraint names 3-5, 7-6
 in cursor names 1-5, 1-6, 1-7, 3-5, 7-6
 in database names 3-6, 6-4, 7-6
 in delimited identifiers 3-10
 in distinct-type names 4-17
 in ESQL filenames 7-4
 in ESQL/C source files 7-6
 in ESQL/COBOL source files 7-6
 in filenames 1-30, 2-23, 3-6, 4-4, 5-4, 6-4, 7-7
 in function names 4-17
 in host variables 1-7, 3-6, 7-6, 8-4
 in index names 3-6, 6-4
 in indicator variables 1-7, 7-6
 in INFORMIX-Universal Server filenames 4-4
 in INFORMIX-Universal Server utilities 4-8
 in literal strings 2-5, 7-7
 in LOAD FROM file 3-52
 in NCHAR columns 3-12
 in numeric formats 7-18
 in NVARCHAR columns 3-15
 in OnLine filenames 2-25, 5-4
 in OnLine utilities 5-8, 6-8
 in opaque-type names 4-17
 in operator-class names 4-18
 in owner names 3-8
 in procedure names 4-18
 in quoted strings 3-33
 in role names 3-6
 in row-type names 4-18
 in SE filenames 2-25, 6-6
 in SE utilities 6-9
 in SQL comments 3-34
 in SQL identifiers 3-5, 4-17
 in statement IDs 1-5, 1-6, 1-7, 3-6, 7-6

in stored procedures 1-5, 1-6, 1-7, 3-6, 3-7
 in synonym names 3-7
 in table names 1-5, 1-6, 1-7, 3-7, 6-4, 7-6
 in triggers 3-7
 in UNLOAD TO file 3-52
 in view names 1-5, 1-6, 1-7, 3-7, 7-6
 partial characters 3-36, 7-25
 processing 2-5, 8-8
 representation of Intro-13
 shifting case of 7-23
 SQL examples Intro-13
 support by C compiler 8-8
 support for 1-3, 1-31
 with CC8BITLEVEL 2-5
 with DBTIME 2-22
 with GLS8BITFSYS 2-23
See also Non-ASCII character.

N

Native Language Support (NLS)
 code-set conversion with 1-45
 collation order 1-12
 establishing a connection 1-34
 information in systables 1-26
 precedence of environment variables 1-25, 1-37, 1-40
 reverting to 5-10
 single-byte data 1-3

NCHAR data type
 code-set conversion 1-48, 4-7
 collation order 1-12, 3-12
 description of 3-11
 difference from CHAR 3-12
 in ESQL/C program 8-6
 in regular expressions 1-5
 inserting into database 8-6
 multibyte characters 3-12
 nonprintable characters 3-13
 performance considerations 3-16
 sizing 3-12
 with numeric values 3-13
 NLS. *See* Native Language Support (NLS).

- Non-ASCII character
 - definition of 1-9
 - examples 1-30
 - filtering 8-8
 - in cast names 4-17
 - in column names 1-5, 1-6, 1-7, 3-5, 7-6
 - in comments 2-5, 7-6
 - in connection names 3-5
 - in constraint names 3-5, 7-6
 - in cursor names 1-5, 1-6, 1-7, 3-5, 7-6
 - in database names 3-6, 6-4, 7-6
 - in delimited identifiers 3-10
 - in distinct-type names 4-17, 4-18
 - in ESQL filenames 7-4
 - in ESQL/C source files 7-6
 - in ESQL/COBOL source files 7-6
 - in filenames 2-23, 3-6, 7-7
 - in host variables 1-7, 3-6, 7-6, 8-4
 - in index names 3-6, 6-4
 - in indicator variables 1-7, 7-6
 - in INFORMIX-Universal Server filenames 4-4
 - in literal strings 2-5, 7-7
 - in LOAD FROM file 3-52
 - in OnLine filenames 2-25, 5-4
 - in opaque-type names 4-17
 - in operator-class names 4-18
 - in owner names 3-8
 - in quoted strings 3-33
 - in role names 3-6
 - in row-type names 4-18
 - in SE filenames 2-25, 6-4
 - in SQL comments 3-34
 - in statement IDs 1-5, 1-6, 1-7, 3-6, 7-6
 - in stored procedures 1-5, 1-6, 1-7, 3-6, 3-7
 - in synonym names 3-7
 - in table names 1-5, 1-6, 1-7, 3-7, 6-4, 7-6
 - in triggers 3-7
 - in UNLOAD TO file 3-52
 - in view names 1-5, 1-6, 1-7, 3-7, 7-6

- processing 2-5, 8-8
- support for 1-31
 - with CC8BITLEVEL 2-5
 - with GLS8BITFSYS 2-23*See also* Multibyte character.
- Non-Gregorian calendar 1-16
- Numeric data
 - currency notation in 7-18
 - decimal separator 1-14, 1-19, 7-19, A-7
 - end-user format 1-14, 1-19, 1-39, A-7
 - ESQL functions 7-17
 - format of A-7
 - locale-specific 1-6
 - negative 1-15, 1-19, A-7
 - positive 1-15, 1-19, A-7
 - thousands separator 1-15, 1-19, 7-19, A-7*See also* Data.
- NUMERIC locale category
 - alternative digits 2-31, 2-38, A-7
 - description of A-4, A-7
 - end-user formats 1-15, A-7
 - in client locale 1-24
 - in default locale A-5
 - in locale source file A-12
 - in server-processing locale 1-40
 - numeric-formatting functions 7-19
- Numeric notation 1-15
- NVARCHAR data type
 - code-set conversion 1-48, 4-7
 - collation order 1-12, 3-14
 - description of 3-13
 - difference from VARCHAR 3-14
 - in ESQL/C program 8-6
 - in regular expressions 1-5
 - inserting into database 8-6
 - multibyte characters 3-15
 - nonprintable characters 3-15
 - performance considerations 3-16
 - sizing 3-14

O

- .o file extension 8-11
- OCTET_LENGTH function 3-43
- onaudit utility 4-10, 5-9
- oncheck utility 4-10, 5-9
- OnLine database server. *See* INFORMIX-OnLine Dynamic Server.
- On-line manuals Intro-15
- onload utility 4-10, 5-9
- onlog utility 4-10, 5-9
- onmode utility 1-7, 5-10
- onshowaudit utility 4-10, 5-9
- onspaces utility 4-10, 5-9
- onstat utility 4-10, 5-9
- onunload utility 4-10, 5-9
- Opaque data type 4-17
- Operating system
 - 8-bit clean 2-24, 4-5, 5-5, 6-4, 7-5
 - character encoding 1-43
 - compatible locale files A-11
 - limitations 7-4
 - locales A-24
 - need for code-set conversion 1-43
 - saving disk space A-20
- Operator class 4-18
- ORDER BY clause (SELECT) 1-10, 3-21
- Owner name 3-8

P

- Partial characters 3-36, 7-25
- Pathname 4-4, 5-4
- Platform icons Intro-8
- Precedence. *See* Environment variable.
- PREPARE statement 3-6
- Printed manuals Intro-16

Q

- Quoted string 3-10, 3-33

R

Radix character. *See* Decimal separator.
Range matches 3-29
rdatestr() library function 1-13, 7-8, 7-9
rdefmtdate() library function 7-8, 7-10
rdownshift() library function 7-23
registry file A-18, A-20
Regular expression 1-5, 1-26
Related reading Intro-17
Relational-operator conditions 3-24
Release notes Intro-17
rfmtdate() library function 7-8, 7-10
rfmtdec() library function 7-17
rfmtdouble() library function 7-17
rfmtlong() library function 7-17, 7-18, 7-20
Role 3-6
Row data type 4-18
rstrdate() library function 7-8, 7-9
rupshift() library function 7-23

S

Sample-code conventions Intro-11
SE database server. *See* INFORMIX-SE database server.
secheck utility 6-9
SELECT statement
 and collation order 1-10
 collation of character data 3-20, 3-21
 embedded 3-50
 LIKE keyword 3-30
 MATCHES relational operator 3-28
 ORDER BY clause 1-10, 3-21
 select-list columns 8-14
 specifying literal matches 3-28, 3-31
 specifying matches with a range 3-29

 specifying quoted strings 3-33
 using length functions 3-40
 using TRIM 3-40, 8-16
 WHERE clause 1-10, 3-24
selog utility 6-9
Server code set 1-41, 1-47, 4-6, 5-6, 6-5, 7-5
Server computer
 server code set 1-41
 setting DB_LOCALE 1-28
 setting SERVER_LOCALE 1-28
Server locale
 code set. *See* Server code set.
 definition of 1-27
 INFORMIX-Universal Server filenames 4-4
 OnLine filenames 5-4
 sample 1-23, 1-27, 1-35
 SE filenames 6-5
 server code set 1-41
 setting 1-28
 uses of 4-3, 5-3, 6-3
 See also SERVER_LOCALE environment variable.
Server-processing locale
 COLLATION category 1-38
 CTYPE category 1-38
 date data 1-39
 definition of 1-34
 determining 1-34
 filename checking 4-7, 5-7, 6-6
 initialization of 1-34
 localized order 1-36
 MESSAGES category 1-40
 MONETARY category 1-40
 monetary data 1-39
 NUMERIC category 1-40
 numeric data 1-39
 precedence of environment variables 1-37, 1-39
 sample 1-35
 TIME category 1-40
 time data 1-39

SERVER_LOCALE environment variable
 default value 1-28
INFORMIX-Universal Server
 filenames 4-4
 location of message files 2-16
 OnLine filenames 5-4
 precedence of 2-16
 role in code-set conversion 1-47
 SE filenames 6-4
 server code set 1-41
 setting 1-28
 syntax 2-41
 See also Server locale.
SET EXPLAIN statement 1-27
Single-byte character Intro-12, 1-3, 1-8, 3-35, 3-38
SMALLFLOAT data type 1-14, A-7
SMALLINT data type 1-14, A-7
Software dependencies Intro-5
Sort order. *See* Collation order.
Spanish locale 1-30
SQL API products
 comments 7-6
 ESQL library enhancements 7-7
 filenames 7-7
 handling non-ASCII characters 7-3
 host variables 7-6
 identifiers 1-31
 literal strings 7-7
 SQL identifier names 7-6
 using GLS8BITFSYS 2-25, 7-5
SQL identifier
 delimited 3-10
 examples 3-9
 multibyte characters 3-5, 4-17
 non-ASCII characters 1-31, 7-6
 owner names 3-8
 rules for 3-4
SQL length function
 CHAR_LENGTH 3-45
 classification of 3-40
 LENGTH 3-40
 OCTET_LENGTH 3-43
 using 3-40

SQL segments 3-7

SQL statement

- CONNECT 3-5
- CREATE CAST 4-17
- CREATE DISTINCT TYPE 4-17
- CREATE FUNCTION 4-17
- CREATE INDEX 3-4, 3-6, 3-20
- CREATE OPAQUE TYPE 4-17
- CREATE OPCLASS 4-18
- CREATE PROCEDURE 3-6, 3-7, 4-18
- CREATE ROLE 3-6
- CREATE ROW TYPE 4-18
- CREATE SYNONYM 3-7
- CREATE TABLE. *See* CREATE TABLE statement.
- CREATE TRIGGER 3-7
- CREATE VIEW 3-7
- data manipulation 3-49
- DECLARE 3-5
- DELETE. *See* DELETE statement.
- DESCRIBE 8-14
- end-user formats in 1-12
- FETCH 3-6
- in code-set conversion 1-48, 4-8
- INSERT. *See* INSERT statement.
- LOAD 3-6, 3-49, 3-52
- PREPARE 3-6
- SET EXPLAIN 1-27
- UNLOAD 3-49, 3-52
- UPDATE. *See* UPDATE statement.

SQL utilities 4-11, 5-10, 6-9

SQLBYTES data type constant 8-13

SQLCA structure

- connection warnings 1-33, 7-25
- SQLWARN7 flag 1-33, 1-34, 1-38, 1-47, 7-25

sqlca structure

- sqlerrm 1-48, 4-8
- sqlwarn.sqlwarn7 7-25

sqlca.sqlwarn.sqlwarn7 flag 7-25

SQLCHAR data type constant 8-17

sqlda structure 8-12, 8-14

sqlda.sqlvar.sqldata field 8-15

sqlda.sqlvar.sqllen field 8-15

sqlda.sqlvar.sqlname field 8-16

SQLNVCCHAR data type

- constant 8-17, 8-18

SQLTEXT data type constant 8-13

sqltypes.h header file 8-13, 8-18

sqlvar_struct structure

- description of 8-15
- sqldata field 8-15
- sqllen field 8-15
- sqlname field 8-16
- storing column data 8-15, 8-16

SQLVCHAR data type

- constant 8-17, 8-18

SQLWARN7 warning flag 1-33, 1-34, 1-38, 1-47, 7-25

Statement identifier 1-5, 1-6, 1-7, 1-48, 3-6, 4-8, 7-6

Stored procedure 1-5, 1-6, 1-7, 1-48, 3-6, 3-7, 4-8

String. *See* Character data; Quoted string; Substring.

Substring 3-34, 3-39

Synonym 3-7

systables system catalog table 1-26

System catalog 1-26, 2-18

T

Table (database)

- in code-set conversion 1-48, 4-8
- in SE database 6-4
- naming 1-5, 1-6, 1-7, 3-7, 6-4, 7-6
- SE names 6-6

Taiwanese dates 1-16, 1-51, 2-12

Territory 1-20, 1-33, A-10

TEXT data type

- code-set conversion 1-48, 4-7
- collation order 1-12
- GLS aspects 3-19
- in code-set conversion 8-12
- partial characters 3-39

See also Binary large object (blob).

Thousands separator 1-15, 1-19, 3-49, 7-19, A-7

Time data

- customizing format of 1-50
- end-user format 1-15, 1-19, 1-39, 1-50, A-9
- format of A-8
- locale-specific 1-6, 1-14
- precedence of environment variables 1-52, 7-17
- with DBTIME 2-20
- with GL_DATE 2-35

See also Data; DATETIME data type.

TIME locale category

- description of A-4, A-8
- end-user formats 1-17, A-9
- era information 2-31, 2-38, A-8
- in client locale 1-24
- in default locale A-5
- in locale source file A-12
- in server-processing locale 1-40

Trigger 3-7

TRIM function 3-40, 8-16

U

UNIX environment

- default code set 1-19
- default locale 1-21
- glfiles utility 1-21
- location of message files 1-49
- location of operating-system locales A-11
- setting environment variables 1-28
- supported code-set conversions 1-46
- supported locales 1-21

Unload file 3-52

UNLOAD statement 3-49, 3-52

UPDATE statement

- embedded SELECT 3-50
- era-based dates 3-50
- GLS considerations 3-49
- SET clause 3-50
- WHERE clause conditions 3-50

User-defined function 4-17

User-defined procedure 4-18

Utility

- chkenv 4-11, 5-10
- database server 1-6
- DB-Access 1-6
- dbaccess 4-10, 5-10, 6-9
- dbexport 1-7, 4-11, 5-10, 6-9
- dbimport 4-11, 5-10, 6-9
- dbload 4-11, 5-10, 6-9
- dbschema 4-11, 5-10, 6-9
- glfiles 1-21, 1-46, A-21
- INFORMIX-Universal Server
 - utilities 4-8
- onaudit 4-10, 5-9
- oncheck 4-10, 5-9
- OnLine utilities 5-8
- onload 4-10, 5-9
- onlog 4-10, 5-9
- onmode 1-7, 5-10
- onshowaudit 4-10, 5-9
- onspaces 4-10, 5-9
- onstat 4-10, 5-9
- onunload 4-10, 5-9
- SE utilities 6-8
- secheck 6-9
- selog 6-9
- SQL utilities 4-11, 5-10, 6-9
- supporting multibyte
 - characters 4-8, 5-8, 6-8, 6-9
- U.S. English locale. *See* Default locale.

V

VARCHAR data type

- code-set conversion 1-48, 4-7
- collation order 1-12
- difference from
 - NVARCHAR 3-14
- GLS aspects 3-18
- performance considerations 3-16

View 1-5, 1-6, 1-7, 3-7, 7-6

W

- Warnings 1-33, 1-34, 1-38, 1-47, 7-24
- WHERE clause
 - and collation order 1-10
 - BETWEEN condition 3-25
 - IN condition 3-27
 - in DELETE statement 3-50
 - in INSERT statement 3-50
 - in UNLOAD statement 3-50
 - in UPDATE statement 3-50
 - logical predicates 3-24
 - relational-operator condition 3-24
- White space
 - defined Intro-14
 - in formatting directives 2-26, 2-28, 2-35
 - in locale 2-26, A-6
 - multibyte Intro-14
 - single-byte Intro-14
 - trailing Intro-14
- Wildcard character 3-31
- Windows environments
 - default code set 1-19
 - default locale 1-21
 - location of message files 1-49
 - location of operating-system
 - locales A-11
 - setting environment
 - variables 1-29
 - supported code-set
 - conversions 1-46
 - supported locales 1-29

X

- X/Open compliance
 - level Intro-18

Symbols

- (minus sign)
 - wildcard in MATCHES
 - clause 3-31
- % (percent)
 - wildcard in LIKE clause 3-31
- * (asterisk)
 - wildcard in MATCHES
 - clause 3-31
- ? (question mark)
 - wildcard in MATCHES
 - clause 3-31
- @ (at sign) 7-20
- [] (brackets)
 - wildcards in MATCHES
 - clause 3-31
- ^ (caret)
 - wildcard in MATCHES
 - clause 3-31
- _ (underscore)
 - wildcard in LIKE clause 3-31