IBM Informix GLS

User's Guide

IBM Informix Extended Parallel Server, Version 8.4
IBM Informix Dynamic Server, Version 9.4
GLS Library, Version 4.0

March 2003
Part No. CT1SNNA

Note:
Before using this information and the product it supports, read the information in the appendix entitled "Notices."

Table of Contents

Chapter 3 SQL Features

Chapter 4 Database Server Features

Introduction

In This Introduction

This introduction provides an overview of the information in this manual and describes the conventions it uses.

About This Manual

This manual describes the Global Language Support (GLS) feature available in IBM Informix products. The GLS feature allows IBM Informix application-programming interfaces (APIs) and IBM 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.

This manual provides GLS information on Informix database servers for both Microsoft Windows and UNIX.

Types of Users

This manual is written for system administrators and application developers who want to use the GLS environment to crerate internationalized database management applications with IBM Informix products.

This manual is primarily intended for those users who need to use IBM Informix products with a nondefault locale. It assumes that you are familiar with Informix database servers and associated products. If that is not the case, refer to your *Getting Started Guide*.

If you need more information about features of your operating system to support non-ASCII characters in filenames, pathnames, and other contexts, see your operating system documentation.

Software Dependencies

This manual assumes that you are using IBM Informix Dynamic Server, Version 9.4, as your database server with Version 4.0 of the GLS library. (IBM Informix Extended Parallel Server, Version 8.4 uses Version 3.13 of the GLS library; see also the Release Notes of your IBM Informix database server, as described in a subsequent section, "Additional Documentation.")

Assumptions About Your Locale

IBM Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a GLS locale.

This manual assumes that you use the U.S. 8859-1 English locale as the default locale. The default is **en_us.8859-1** (ISO 8859-1) on UNIX platforms or **en_us.1252** (Microsoft 1252) for Windows environments. This locale supports U.S. English format conventions for dates, times, and currency, and also supports the ISO 8859-1 or Microsoft 1252 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ. If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

Demonstration Databases

The DB-Access utility, which is provided with your IBM Informix database server products, includes one or more of these demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix manuals are based on this database.
- XPS — The **sales_demo** database illustrates a dimensional schema for data warehousing applications. For conceptual information about dimensional data modeling, see the *IBM Informix Database Design and Implementation Guide*. ♦
- IDS — The **superstores_demo** database illustrates an object-relational schema. This database contains examples of extended data types, type and table inheritance, and user-defined routines. ♦

For information about how to create and populate the demonstration databases, see the *IBM Informix DB-Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX platforms and in the **%INFORMIXDIR%\bin** directory in Windows environments.

New Features of Dynamic Server

The SET COLLATION statement can specify the localized collating order of a locale that is different from what **DB_LOCALE** specifies. This localized collation is applied to sorting operations on NCHAR and NVARCHAR data types in the same session. Database objects that sort NCHAR or NVARCHAR values use the collating order that was in effect when the object was created, if this differs from the **DB_LOCALE** setting.

Version 4.0 of the GLS Library, which is provided with Version 9.40 of Dynamic Server, provides the following additional new features:

- The open-source International Components for Unicode (ICU) implementation of the Unicode code set (UTF-8)
- Unicode collation, based on the ICU Unicode Collation Algorithm, in locales that use the Unicode code set
- Full support of the GB18030-2000 code set for Chinese characters

(For a comprehensive list of new Dynamic Server features in Version 9.40, see the *Getting Started Guide*.)

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set. The following conventions are discussed:

- Typographical conventions
- Icon conventions

- Syntax conventions
- Command-line conventions
- Example-code conventions
- Character-representation conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics ***italics*** *italics*	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
monospace *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 one or more product- or platform-specific paragraphs.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

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 three types of information, as the following table describes. This information always appears in italics.

Icon	Label	Description
	***Warning:***	Identifies paragraphs that contain vital instructions, cautions, or critical information
	***Important:***	Identifies paragraphs that contain significant information about the feature or operation that is being described
	***Tip:***	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described

Feature, Product, and Platform Icons

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

Icon	Description
DB API	Identifies information that is specific to the DataBlade API
E/C	Identifies information that is specific to IBM Informix ESQL/C
IDS	Identifies information that is specific to IBM Informix Dynamic Server

(1 of 2)

Icon	Description
UNIX	Identifies information that is specific to UNIX platforms
Windows	Identifies information that is specific to Windows environments
XPS	Identifies information or syntax that is specific to IBM Informix Extended Parallel Server

(2 of 2)

These icons can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-, product-, or platform-specific information that appears within one or more paragraphs within a section.

Compliance Icons

Compliance icons indicate paragraphs that provide guidelines for complying with a standard.

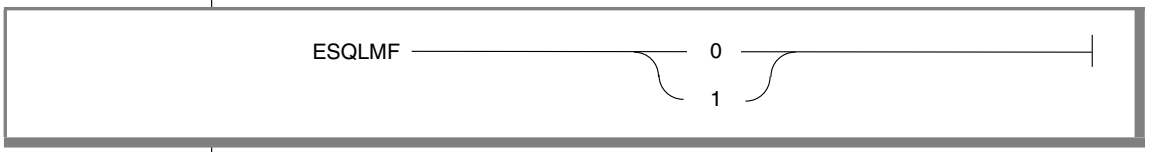
Icon	Description
ANSI	Identifies information that is specific to an ANSI-compliant database
+	Identifies information that is an Informix extension to ANSI SQL-92 entry-level standard SQL

These icons can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-, product-, or platform-specific information that appears within one or more paragraphs within a section.

Syntax Conventions

This section describes conventions for syntax diagrams. Each diagram displays the sequences of required and optional keywords, terms, and symbols that are valid in a given statement or segment, as Figure 1 shows.

Figure 1
Example of a Simple Syntax Diagram



Each syntax diagram begins at the upper-left corner and ends at the upper-right corner with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement.

Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. The path always approaches elements from the left and continues to the right, except in the case of separators in loops. For separators in loops, the path approaches counterclockwise. Unless otherwise noted, at least one blank character separates syntax elements.

Elements That Can Appear on the Path

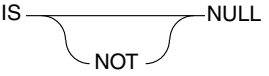
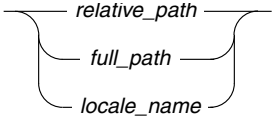
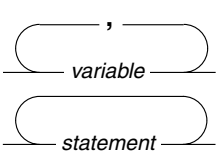
You might encounter one or more of the following elements on a path.

Element	Description
KEYWORD	A word in UPPERCASE letters is a keyword. You must spell the word exactly as shown; however, you can use either uppercase or lowercase letters.
(. , ; @ + * - /)	Punctuation and other nonalphanumeric characters are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.

(1 of 3)

Element	Description
variable	A word in italics represents a value that you must supply. A table immediately following the diagram explains the value.
Format Qualifiers for Reads p. 2-23 Format Qualifiers for Reads	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.
Back to GL_DATE p. 2-17	A reference in a box in the upper-right corner of a subdiagram refers to the next higher-level diagram of which this subdiagram is a member.
E/C	An icon is a warning that this path is valid only for some products, or only under certain conditions. Characters on the icons indicate what products or conditions support the path. These icons might appear in a syntax diagram: XPS — This path is valid only for IBM Informix Extended Parallel Server. DB — This path is valid only for DB-Access. E/C — This path is valid only for IBM Informix ESQL/C. IDS — This path is valid only for IBM Informix Dynamic Server.
ALL	A shaded option is the default action.
▶ · · · ▶	Syntax delimited by a pair of arrows is a subdiagram. (But ellipses never appear within syntax diagrams.)

(2 of 3)

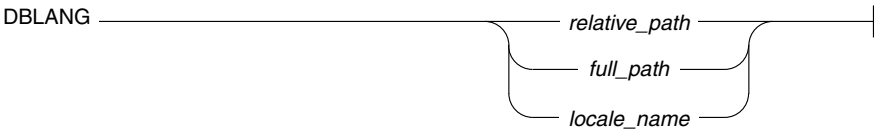
Element	Description
	The vertical line terminates the syntax diagram. Diagrams with this terminator are not subdiagrams.
	A branch above or below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
	A set of multiple branches indicates that a choice among more than two different paths is available. (Note that the underscore (_) symbols here are part of the name of the term, rather than part of the path.)
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items. If no symbol appears, a blank space is the default separator. In most contexts, a *statement* loop can be empty. (This is an exception to the convention that any term on the main part is required.)

(3 of 3)

How to Read a Syntax Diagram

Figure 2 shows a syntax diagram that uses some of the path elements that the previous table lists.

Figure 2
Example of a Syntax Diagram



To use this diagram to specify a valid setting for this environment variable, start at the top left with the keyword **DBLANG**. Then follow the diagram to the right, proceeding through the options that you want.

Figure 2 illustrates the following steps:

1. Type DBLANG.
2. You must specify a *pathname* or a *locale*. Type the relative path, full path, or locale name, as you desire.
3. Follow the diagram to the terminator.

   Your **DBLANG** specification is complete.

Command-Line Conventions

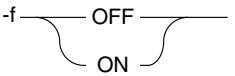
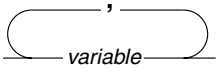
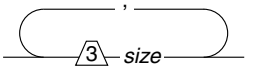
This section defines and illustrates the format of commands that are available in IBM Informix products. These commands have their own conventions, which might include alternative forms of a command, required and optional parts of the command, and so forth.

Each diagram displays the series 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 reverse direction. Each path describes a valid form of the command. You must supply a value for terms that appear 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 IBM Informix product. It might appear alone or precede one or more options. You must spell a command exactly as shown and use lowercase letters.
variable	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.

(1 of 2)

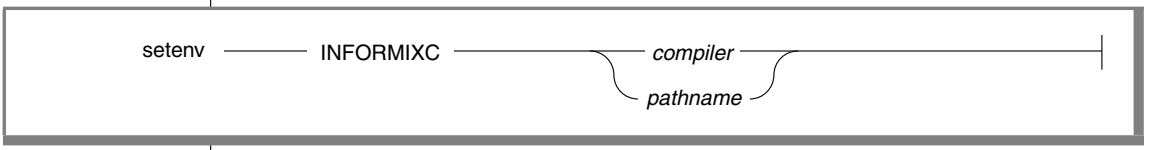
Element	Description
-flag	A flag is usually an abbreviation for a function, menu, or option name, or for a compiler or preprocessor argument. You must enter a flag exactly as shown, including the preceding hyphen.
.ext	A filename extension, such as **.sql** or **.cob**, might follow a variable that represents 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.
	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.
	A shaded option is the default action.
	Syntax within a pair of arrows indicates a subdiagram.
	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 (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. You can specify *size* no more than three times within this statement segment.

(2 of 2)

How to Read a Command-Line Diagram

Figure 3 shows a command-line diagram that uses some of the elements that are listed in the previous table.

Figure 3
Example of a Command-Line Diagram



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

Figure 3 illustrates the following steps:

1. Type `setenv`.
2. Type `INFORMIXC`.
3. Supply either a compiler name or a pathname.

   After you choose *compiler* or *pathname*, you come to the terminator. Your command is complete.
4. Press RETURN to execute the command.

Example-Code Conventions

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

```
CONNECT TO stores_demo
...
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 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. Except within quoted strings, literal ellipses are not valid in SQL.*

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 are displayed. Multibyte characters are usually ideographic (such as Japanese or Chinese characters), but this manual does not depict the actual multibyte characters. Instead, it uses ASCII characters to represent both single-byte and multibyte characters. This section describes how this manual represents multibyte and single-byte characters abstractly.

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
```

Here `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
```

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

```
abcdef
```

Tip: *The letter "s" does not appear in examples that represent strings of single-byte characters. The manual reserves the letter "s" as a symbol to represent a single-byte whitespace character. See also* *"Whitespace Characters in Strings" on page 17*.

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:

$A^1 . . . A^n$

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^1A^2

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

$A1A^2A^3A^4$

The next 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 about using multibyte characters in SQL identifiers, see "Naming Database Objects" on page 3-3.

Single-Byte and Multibyte Characters in the Same String

For a multibyte code set, a given string might be composed of both single-byte and multibyte characters. To represent mixed strings, this manual simply combines the formats for multibyte and single-byte characters. The next example represents a string with four characters, where the first and fourth characters are single-byte characters, and the second and third characters are multibyte characters that consist of two bytes each:

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

Whitespace Characters in Strings

Whitespace is a series of one or more characters that display as blank space, Each GLS locale defines what characters are whitespace characters.

For example, both the TAB (ASCII 9) and blank space (ASCII 32) might be defined as whitespace characters in one locale, but certain combinations of the CTRL key and another character might be defined as whitespace characters in a different locale.

The convention for representing a single-byte whitespace in this manual is the letter "s." The following notation represents one single-byte whitespace:

```
s
```

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

```
ss
```

The following notation represents a multibyte whitespace character:

$s^1 \ldots s^n$

Here s^1 represents the first byte of the whitespace character, and s^n represents the last byte of the whitespace character, where *n* can range between two and four. The following notation represents one 4-byte whitespace character:

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

Trailing Whitespace Characters

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

```
abcss
```

The next example represents a string of five characters (three characters of data and two trailing whitespace characters) in a multibyte code set where each of the data characters and whitespace characters consists of two bytes:

$A^1 A^2 B^1 B^2 C^1 C^2 s^1 s^2 s^1 s^2$

In some locales, a string can contain both single-byte and multibyte whitespace characters. For example, consider the following string:

```
abcss1s2sss1s2
```

The string has three single-byte characters (abc), a single-byte whitespace character (s), a multibyte whitespace character (s1s2), two single-byte white-space characters (ss), and one multibyte whitespace character (s1s2):

Additional Documentation

Database server documentation is provided in a variety of formats:

- **Online manuals.** The documentation CD in your media pack allows you to print the product documentation. You can obtain the same online manuals at the IBM Informix Online Documentation site at **http://www-3.ibm.com/software/data/informix/pubs/library/**.
- **Online help.** This facility provides context-sensitive help, an error message reference, language syntax, and more.
- **Documentation notes and release notes.** Documentation notes, which contain additions and corrections to the manuals, and release notes, which describe the database server, are located in the directory where the product is installed. Please examine these files, which contain vital information about application and performance issues.

UNIX

On UNIX platforms that use the default locale, the following online files appear in the **$INFORMIXDIR/release/en_us/0333** directory.

Online File	Description
ids_gls_docnotes_9.40.html *or* **xps_gls_docnotes_9.40.html**	The documentation notes file for your version of this manual describes topics that are not covered in the manual or that were modified since publication.
ids_unix_release_notes_9.40.html, **ids_unix_release_notes_9.40.txt** (*for* Dynamic Server) *or* **xps_release_notes_8.40.html** **xps_release_notes_8.40.txt** (*for* Extended Parallel Server)	The release notes file describes feature differences from earlier versions of Dynamic Server and how these differences might affect your products. This file also contains information about any known problems and their workarounds. (The **.txt** version is in ASCII text format; the **.html** version can be displayed with a browser.)
ids_machine_notes_9.40.txt (*for* Dynamic Server) *or* **xps_machine_notes_8.40.txt** (*for* Extended Parallel Server)	The machine notes file describes any special actions that you must take to configure and use IBM Informix products on your computer. Machine notes are named for the product described.

♦

Windows

The following items appear in the **Informix** folder. To display this folder, choose **Start→Programs→Informix→Documentation Notes or Release Notes** from the task bar.

Program Group Item	Description
Documentation Notes	This item includes additions or corrections to manuals with information about features that might not be covered in the manuals or that have been modified since publication.
Release Notes	This item describes feature differences from earlier versions of IBM Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.

Machine notes do not apply to Windows platforms. ♦

- **Error message files**. IBM Informix software products provide ASCII files that contain error messages and their corrective actions.

  UNIX

  To read the error messages on UNIX, you can use the **finderr** command to display the error messages online. ♦

  Windows

  To read error messages and corrective actions on Windows, use the **Informix Error Messages** utility. To display this utility, choose **Start→Programs→Informix** from the task bar. ♦

Related Reading

For a list of publications that provide an introduction to database servers and operating-system platforms, refer to your *Getting Started Guide*.

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. IBM 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. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

IBM Welcomes Your Comments

We want to know about any corrections or clarifications that you would find useful in our manuals that would help us with future versions. 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

Send electronic mail to us at the following address:

docinf@us.ibm.com

This address is reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact Customer Services.

We appreciate your suggestions.

Chapter

1

GLS Fundamentals

In This Chapter

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

The GLS feature lets you create databases using the diacritics, collating sequence, and monetary and time conventions of the language that you select. No ONCONFIG configuration parameters exist for GLS, but you must set the appropriate environment variables.

This chapter introduces basic concepts and describes the GLS feature. It includes the following sections:

Using the GLS Feature

In a database application, some of the tasks that the database server and the client application perform depend on the language and culture conventions of the data that they handle. For example, the database server must sort U.S. English data differently from Korean character data. The client application must display Canadian currency differently from Thai currency.

If the Informix database server or client 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.

With support for GLS, IBM Informix products no longer need to specify how to process culture-specific information directly. Culture-specific information resides in a GLS locale. When an IBM Informix product needs culture-specific information, it calls the GLS library, which accesses the GLS locale and returns the information to the IBM Informix product.

The GLS feature is a portable way to support culture-specific information. Although many operating systems provide support for non-English data, 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.

The GLS feature can access culture-specific information on a UNIX or Windows operating system. IBM Informix products can locate the locale information on any platform to which they are ported.

Windows

In order for GLS to support a nondefault locale, the version of Windows that you are using must also support that locale. That is, you cannot support a Japanese client application on Windows unless that application is running on the Japanese version of Windows. ♦

To use the GLS feature, the tasks that you need to perform depend on whether you are a system administrator, database administrator, end user of a client application, end user of a database server utility, or client application developer. The following table lists these optional and mandatory tasks.

Audience	Optional Tasks	Mandatory Tasks
System administrator, database administrator, or end user of client application	■ For non-default locales, set the **DB_LOCALE**, **CLIENT_LOCALE**, and **SERVER_LOCALE** environment variables. ■ To customize end-user formats, set the **GL_DATE**, **GL_DATETIME**, and **DBMONEY** environment variables. For ESQL/C, you can set **DBTIME** instead of **GL_DATETIME**. ■ To configure a GLS environment for ESQL/C, set the **CC8BITLEVEL** and **ESQLMF** environment variables. ■ To perform additional configuration for the GLS environment, set the **DBLANG** and **GLS8BITFSYS** environment variables. ■ To issue an SQL statement, follow the guidelines in Chapter 3, "SQL Features," and Chapter 4, "Database Server Features." ■ To remove GLS files, follow the guidelines in "Removing Unused Files" on page A-15. ■ To get information about GLS files on UNIX, follow the guidelines in "The glfiles Utility" on page A-16.	None
End user of database server utility	Same as above	Follow the guidelines in "Locale-Specific Support for Utilities" on page 4-7.
Client application developer	■ Same as above ■ To develop an internationalized client application, follow the guidelines in "Internationalizing Client Applications" on page 5-7 and the *IBM Informix GLS Programmer's Manual*.	■ Follow the guidelines in Chapter 5, "General SQL API Features." ■ For an ESQL/C application, also follow the guidelines in Chapter 6, "IBM Informix ESQL/C Features."

GLS Support by IBM Informix Products

GLS support is provided for these IBM Informix products and utilities:

- Informix database servers
- IBM Informix client applications and database server utilities
- The IBM Informix GLS application programming interface

Sections that follow outline the features that GLS support provides for the first two types of IBM Informix products.

For information about how to migrate a database server whose databases contain non-English data, see the *IBM Informix Migration Guide*.

Informix Database Servers

GLS was introduced in IBM Informix OnLine Dynamic Server. Previously, ALS language support was provided for non-English databases with Asian (multibyte) characters and NLS language support for non-English databases with single-byte characters. GLS is a single feature that provides support for single-byte and multibyte data in non-English languages. For backward compatibility, GLS products also support all of the NLS environment variables and a subset of the ALS environment variables. For a list of these variables, see the *IBM Informix Migration Guide*.

Culture-Specific Features

With the GLS feature, Informix database servers 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 SPL routines, and you can use a collation order that suits local customs.

  You can also use non-ASCII characters in many other contexts. For example, you can use them to specify 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 expressions

  You can use non-ASCII characters in 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 specific 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 "Performing Code-Set Conversion" on page 1-40.

IBM Informix Client Applications and Utilities

In general, a client application is a program that runs on a workstation or a PC on a network. To the GLS feature, a *client application* can be either an IBM Informix SQL API product (such as IBM Informix ESQL/C) or an Informix database server utility (such as DB-Access, **dbexport**, or **onmode**). These IBM Informix client applications provide support for the GLS feature:

- The DB-Access utility, which is provided with Informix database servers, allows user-specifiable database objects such as tables, columns, views, statements, cursors, and SPL routines to include non-ASCII characters and to be sorted according to localized collation rules. For more information on identifiers, see "Non-ASCII Characters in Identifiers" on page 3-5. For general information about DB-Access, refer to the *IBM Informix DB-Access User's Guide*.

- 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 SPL routines to include non-ASCII characters. For more information, see Chapter 5, "General SQL API Features."

- Database server utilities such as **dbexport** or **onmode** allow many command-line arguments to include non-ASCII characters. For more information, see Chapter 4, "Database Server Features."
- GLS is also a feature of IBM Informix Dynamic 4GL (Version 3.0 and higher), IBM Informix 4GL (Version 7.2 and higher), and IBM Informix SQL (Version 7.2 and higher). For details of GLS implementation, refer to the documentation of these IBM Informix products.

The IBM Informix GLS Application Programming Interface

IBM Informix GLS is an application programming interface (API) that lets DataBlade module developers and ESQL/C programmers develop internationalized applications with a C-language interface.

The macros and functions of IBM Informix GLS provide access within an application to GLS locales, which contain culture-specific information. You can use IBM Informix GLS to write programs (or change existing programs) to handle different languages, cultural conventions, and code sets.

All IBM Informix GLS functions access the *current processing locale*, which is the locale that is currently in effect for an application. It is based on either the client locale (for ESQL/C client applications and client LIBMI applications) or the server-processing locale (for DataBlade user-defined routines).

IBM Informix GLS provides macros and functions to help you perform the following internationalization tasks:

- Process single-byte, multibyte, and wide characters
- Process single-byte, multibyte, and wide-character strings
- Memory management for multibyte and wide-character strings
- Convert date, time, money, and number strings to and from binary values
- Process input and output multibyte-character streams

IBM Informix client applications as well as database servers can access IBM Informix GLS. For applications, you link the IBM Informix GLS library to your application to perform locale-related tasks. Informix database servers automatically include the IBM Informix GLS library. For more information, see the *IBM Informix GLS Programmer's Manual*.

Supported Data Types

The GLS feature supports the following data types:

- SQL character data types
  - CHAR, VARCHAR, NCHAR, and NVARCHAR
  - IDS LVARCHAR ♦
  - TEXT and BYTE

  For information about GLS considerations for the character data types, see "Using Character Data Types" on page 3-11.
- SQL number and MONEY data types

  For information about GLS considerations for number and MONEY data types, see "Numeric and Monetary Formats" on page 1-19.
- SQL DATE, and DATETIME data types

  For information about GLS considerations for DATE, and DATETIME data types, see "Date and Time Formats" on page 1-20.
- IDS User-defined data types
  - Opaque data types
  - Complex data types
  - Distinct data types
- Smart large objects
  - BLOB
  - CLOB

  For GLS considerations regarding user-defined data types and smart large objects, see "Handling Extended Data Types" on page 3-51. ♦
- ESQL/C character data types
  - **char**
  - **fixchar**
  - **string**
  - **varchar**
  - **lvarchar**

  For information about ESQL/C data types, see the *IBM Informix ESQL/C Programmer's Manual*.

Additional GLS Support

IBM Informix products include a core set of GLS locale files, including the default locale and most locales to support English, Western European, Eastern European, Asian, and Arabic territories. If you do not find a locale to support your language and territory, you can get additional locales in the International Language Supplement (ILS) product, which provides all available GLS locales and code-set conversion files. It also includes error messages to support several European languages.

For more information about available GLS locales and IBM Informix Language Supplements, contact your sales representative. For more information about how to create customized message files, see "Locating Message Files" on page 1-45.

Understanding 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. A GLS locale is a set of Informix files that bring together the information about data that is specific to a given culture, language, or territory. In particular, a GLS locale can specify the following:

- The name of the code set that the application data uses
- The classification of the characters in the code set
- The collation (sorting) sequence to use for character data
- The end user format for monetary, numeric, date and time data

IBM Informix products use the following GLS files to obtain locale-related information. For more information, see Appendix A, "Managing GLS Files."

Type of GLS File	Description
GLS locale files	Specify language, territory, writing direction, and other cultural conventions.
Code-set files	Specify how to map each logical character in a character set to a unique bit pattern.
Code-set-conversion files	Specify how to map each character in a "source" code set to a corresponding characters in a "target" code set.
The **registry** file	Associates code-set names and aliases with code-set numbers that specify filenames of locale files and code-set conversion files.

Each database is limited to a single locale, but different databases of the same database server can support different locales.

A single database can store character data from two or more languages that require different character sets by using the open-source International Components for Unicode (ICU) implementation of the Unicode code set (UTF-8). This code set is available in GLS database server locales for many languages and territories. (Locales for some client-side systems also support the ICU code set UTF-8, as well as the ICU code sets UTF-16 and UTF-32.)

IDS

The SET COLLATION statement of Dynamic Server supports more than one localized collating order to sort NCHAR and NVARCHAR character strings. ♦

Code Sets for Character Data

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, Windows Code Page 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 IBM 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 code set or Windows Code Page 1252 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, see "GLS8BITFSYS" on page 2-13.

- 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. Some East-Asian locales support character sets that can contain thousands of ideographic characters; GLS provides full support, for example, for the unified Chinese GB18030-2000 code set, which contains nearly 1.4 million code points. Such languages have code sets that include both single-byte and multibyte characters. These code sets are called *multibyte* code sets.

Some characters in the Japanese SJIS code set, for another example, are 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.*

IBM 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 1-32.

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, see "Character-Representation Conventions" on page 15 of the Introduction.*

Character Classes of the Code Set

A GLS locale groups the characters of a code set into *character classes*. Each class contains characters that have a related purpose. GLS supports 12 classes. The contents of a character class can be language specific. For example, the *lower* class contains all alphabetic lowercase characters in a code set. The code set of the default locale groups the letters `a` through `z` into the lower class, which also includes other lowercase characters such as `á`, `è`, `î`, `õ`, and `ü`.

To be internationalized, your application must not assume which characters belong in a given character class. Instead, use IBM Informix GLS library functions to identify the class of a particular character. For information about the IBM Informix GLS functions to use and a list of character classes and what characters each class contains, see the *IBM Informix GLS Programmer's Manual*.

Collation Order for Character Data

Collation is the process of sorting character strings according to some order. The database server or the client application can perform collation.

The collating order affects the following tasks in SQL SELECT statements:

- 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 the database locale can affect the SELECT statement, see "Collation Order in SELECT Statements" on page 3-30.

Informix database servers support two collation methods:

- Code-set order (the first-to-last order of characters in the code set)
- Localized order (if the locale defines a localized order)

Code-Set Order

Code-set order refers to the order of characters within a code set. The order of the code points in the code set determines the collating 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. But because a=97 and M=77, the string abc sorts after Me, which is not always the preferred result.

The database server uses code-set order to sort columns of these data types:

IDS

- CHAR
- LVARCHAR ♦
- VARCHAR
- TEXT

All code sets that IBM 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 values of these data types, 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 data sorted in code-set order, see Figure 3-2 on page 3-31.

Localized Order

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

The database server uses code-set order to sort columns of these data types:

- NCHAR
- NVARCHAR

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 in the localized order, then the strings Arizona, ARIZONA, and arizona are collated together, as if all three strings were the same string.

Tip: *The COLLATION category of the locale file specifies the localized order, if one exists. For more information, see* *"The COLLATION Category" on page A-5*.

A localized order can also specify a specific type of collation. For example, a telephone book might require the following sort order:

```
Mabin
McDonald
MacDonald
Madden
```

A dictionary, however, might use this collating order for the same names:

```
Mabin
Madden
MacDonald
McDonald
```

If the GLS locale defines a localized order, the database server sorts data from NCHAR and NVARCHAR columns in this localized order. For an example of data sorted in a localized order, see Figure 3-3 on page 3-32.

IDS

Dynamic Server supports the SET COLLATION statement, which can specify a localized collation different from the **DB_LOCALE** setting. The scope of the non-default collating order is the current session, but database objects that perform collation, such as indexes or triggers, use the collating order from the time of their creation when they sort NCHAR or NVARCHAR values. ♦

Unicode Collation

The open-source International Components for Unicode (ICU) implementation of the Unicode code set (UTF-8) is available in GLS locales for many languages and territories. For example, the **en_us.utf8** locale supports the Unicode code set. For more information about ICU, see the ICU website at http://oss.software.ibm.com/icu/.

GLS locales that use the Unicode code set (UIF-8) support Unicode collation of NCHAR and NVARCHAR data by the ICU Unicode Collation Algorithm. For more information about this algorithm, see the Unicode website at http://www.unicode.org/unicode/reports/tr10.

Collation Support

Collation by Informix database servers depends on the data type of the database column. The following table summarizes the collation rules.

Column Data Types	Collating Order
CHAR, VARCHAR, TEXT	Code-set order
LVARCHAR (IDS only)	Code-set order
NCHAR, NVARCHAR	Localized order

The difference in collation is the only distinction between the CHAR and NCHAR data types and between the VARCHAR and NVARCHAR data types. For more information about collation, see "Using Character Data Types" on page 3-11. If a locale does not define a localized order, the database server collates NCHAR and NVARCHAR data values in code-set order.

Important: *In an exception to the general rule that CHAR, LVARCHAR and VCHAR values are always sorted in the code-set order. The MATCHES operator always uses the localized order, if one is defined, to evaluate range expressions for character values, regardless of the data type. See "MATCHES Condition" on page 3-38.*

End-User Formats

The *end-user format* is the format in which a data value is displayed or entered in a client application as a literal string or a character variable.

An end-user format is useful for a data type whose format in the database is different from the format to which users are accustomed. The database server stores data for DATE, DATETIME, MONEY, and numeric data types in compact internal formats within the database.

For example, the database server stores a DATE value as an integer number of days since December 31, 1899, so the date `03/19/96` is `35142`. This internal format is not intuitive.

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

- When an IBM Informix product reads 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 that DB-Access has the default (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 it 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 corresponding internal format (`35142`).

- When an IBM 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 that 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 "Using Enhanced ESQL/C Library Functions" on page 6-11.

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

- Representation of currency notation and numeric format
- Representation of dates and of time-of-day values

You can specify number, currency, date, and time values in an end-user format that is specific to a given country or culture.

Important: *End-user formats of date, time, number, and monetary values do not affect the internal format of the corresponding data types in the database. They affect only how the client application displays the data and interprets data entry.*

The following table lists the values that define the end-user format for each data type that uses end-user formats. For information about the environment variables, see Chapter 2, "GLS Environment Variables." For information about the locale categories, see Appendix A, "Managing GLS Files."

Data Types	Environment Variables	Locale Category
DATE	**GL_DATE**	TIME
DATETIME INTERVAL	**GL_DATE** **GL_DATETIME**	TIME
MONEY	**DBMONEY**	MONETARY
Number (DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INT8, INTEGER, NUMERIC, REAL, SMALL-FLOAT, SMALLINT)	None	NUMERIC

Numeric and Monetary Formats

When an IBM Informix product reads a string that contains numeric or monetary data, it uses the end-user format to determine how to convert this string to the internal value for the database column. When an IBM Informix product prints a string that contains numeric or monetary data, it uses the end-user format to determine how to format the internal value for the database column as a string.

End-user formats for numbers and currency specify these elements:

- The *decimal-separator symbol* 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 locale such as French, the comma is the decimal separator (3,01).
- The *thousands-separator symbol* can appear 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 characters that indicate positive and negative numbers
- The number of digits to group between each appearance of a non-monetary thousands separator.

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

In addition to this notation, monetary data also uses a *currency symbol* to identify the currency unit. This can 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*.

Date and Time Formats

When an IBM Informix product reads a string that contains time data, it uses the end-user format to determine how to convert this string to the internal integer value for a DATETIME column. When an IBM 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, IBM Informix products use the date end-user format to read and print strings for the internal values of the date data types.

Important: *End-user formats specify how client applications view data, but do not affect the internal format of DATETIME or DATE values stored in the database.*

The end-user formats for date and time can include the names and abbreviations for days of the week and months of the year, and the commonly used representations for dates, time (12-hour and 24-hour), and DATETIME values.

End-user formats can include 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 Gregorian calendar year. 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.

Gregorian 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.

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

Here A^1A^2 and B^1B^2 represent multibyte Japanese characters. For more information, see "Customizing Date and Time End-User Formats" on page 1-46.

Setting a GLS Locale

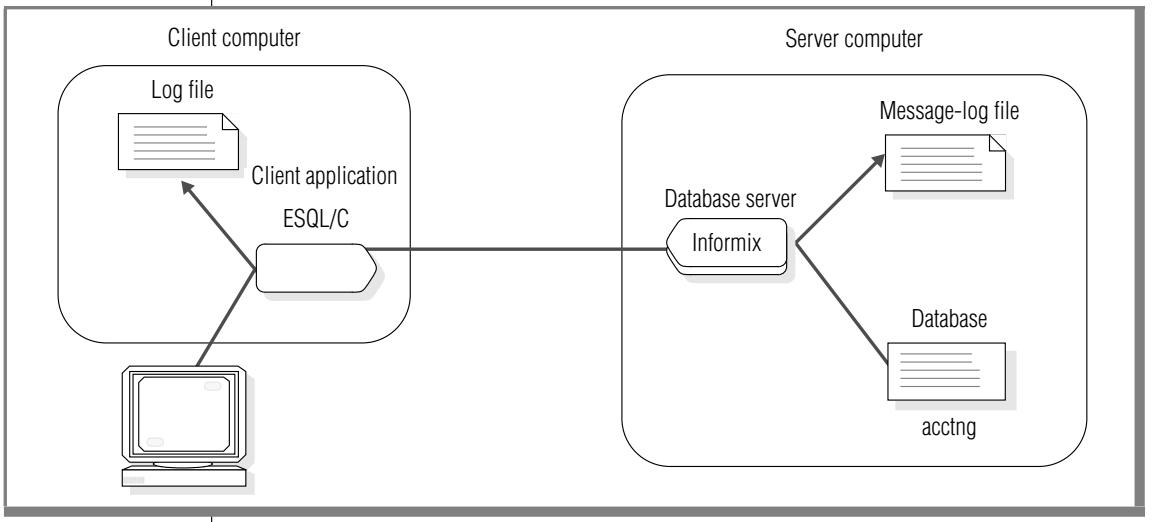
For the database server and the client application to communicate successfully, you must establish the appropriate GLS locales for your environment. A GLS locale name identifies the language, territory, and code set that you want your IBM Informix product to use. For the syntax of the components of locale names, see "CLIENT_LOCALE" on page 2-5.

IBM Informix product use the locale name to find the corresponding *locale files*. 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 **INFORMIXDIR** specifies) called **gls**. For more information on GLS locale files, see Appendix A, "Managing GLS Files."

Locales in the Client/Server Environment

In a client/server environment, the client application, database server, and one or more databases might reside on different computers. Figure 1-1 shows an example of database server connections between an ESQL/C client application and the **acctng** database through an Informix database server.

Figure 1-1
Example of a Client/Server Environment

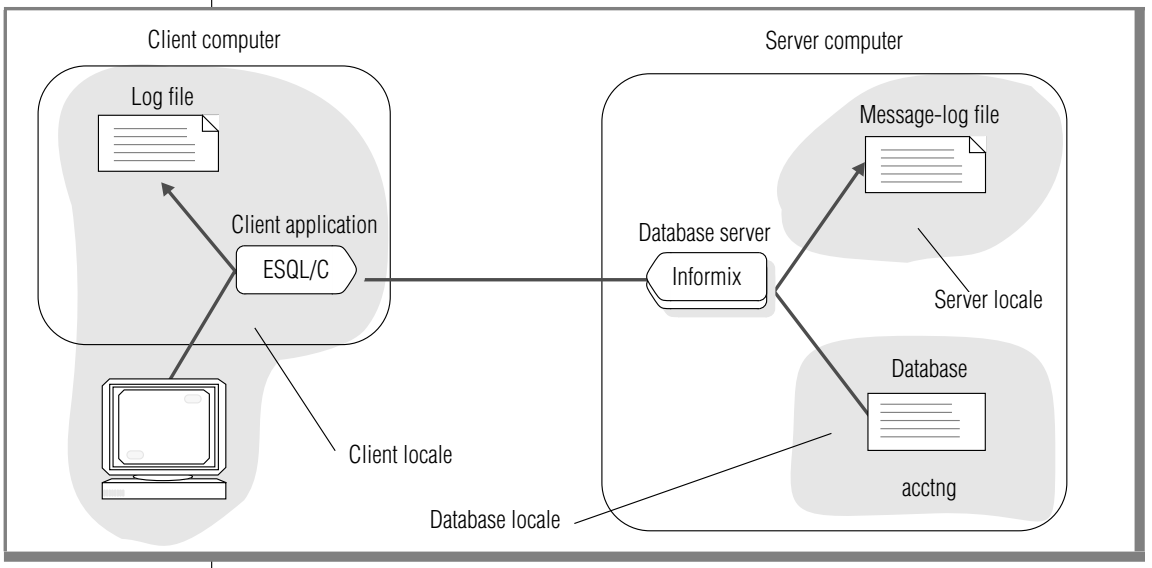


These computers might have different operating systems or different language support. To ensure that these three parts of the database application communicate locale information successfully, IBM Informix products support the following locales:

- 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 example 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.

Sections that follow 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. 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 (ESQL) statements, and host variables.

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

- When the preprocessor for ESQL/C processes a source file, it accepts C 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 "Generating Non-ASCII Filenames" on page 6-6.
- When an ESQL/C 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 "Handling Non-ASCII Characters" on page 6-3.
- 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 1-33.
- When database utilities create files, the filenames and file contents are in the code set that **CLIENT_LOCALE** specifies.
- When a client application looks for product-specific message files, it checks the message directory associated with the client locale.

  For more information, see "Locating Message Files" on page 1-45.

In the example connection that Figure 1-2 on page 1-23 shows, if the client locale is German with the Windows Code Page 1252 (**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, 02/11/1997:

```
Tu., 11. Feb 1997
```

- 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-3.*

To determine the client locale, 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. The default locale (U.S. English)

IDS

Client applications that are based on IBM Informix Dynamic Server use the precedence of steps 2, 3, and 4 in the preceding list. You do not need to set the other environment variables for Dynamic Server client applications. ♦

Support for **DBDATE** and **DBTIME** provides backward compatibility for client applications based on earlier versions of IBM Informix products. We recommend that you use **GL_DATE** and **GL_DATETIME** for new applications.

The Database Locale

The *database locale*, which is set with the **DB_LOCALE** environment variable, specifies the language, territory, and code set that the database server needs to correctly interpret locale-sensitive data types (NCHAR and NVARCHAR) in a particular database. The code set specified in **DB_LOCALE** determines which characters are valid in any character column, as well as the names of database objects such as databases, tables, columns, and views. For more information, see "Naming Database Objects" on page 3-3.

The database locale also specifies the *writing direction*. Most languages are written left-to-right, but some are written right-to-left or top-to-bottom.

IBM 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 "Establishing a Database Connection" on page 1-33.

- 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 the database server accesses columns of locale-sensitive data types, it uses the locale that **DB_LOCALE** specifies. For more information, see "Verifying the Database Locale" on page 1-34.

- When the database server creates a new database, it examines the database locale (**DB_LOCALE)** to determine how to store character information in the system catalog of the database. This information includes operations such as how to handle regular expressions, compare character strings, and ensure proper use of code sets.

The database server stores a condensed version of the database locale in the **systables** system catalog table.

When the database server stores the database locale information directly in the system catalog, it permanently attaches the locale to the database. This information is used throughout the lifetime of the database. In this way, the database server can always determine the locale that it needs to interpret the locale-sensitive data correctly.

IDS

The SET COLLATION statement can specify the localized collation of a different locale to sort NCHAR and NVARCHAR data in the current session. ♦

The condensed version of the database locale is stored in the following two rows of **systables**, which store the condensed locale name in the **site** column:

- The row with **tabid** `90` stores the COLLATION category of the database locale. The collation order determines the order in which the characters of the code set collate. If the database locale defines only a code-set order for collation (as does the default locale, U.S. English), the database server creates CHAR and VARCHAR columns to store the character information. If the database locale defines a localized order for collation, however, the database server creates NCHAR and NVARCHAR columns to store this character information.The **tabname** value for this row is GLS_COLLATE.
- The row with **tabid** `91` stores the CTYPE category of the database locale. The CTYPE category of a locale determines how characters of the code set are classified. The database server uses character classification for case conversion and some regular-expression evaluation.The **tabname** value for this row is GLS_CTYPE.

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

In the 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 this locale is German with the Windows Code Page 1252 (**de_de.1252**), the database server uses a localized order that sorts accented characters, such as ö, after their unaccented counterparts. Thus, the string `öff` sorts after `ord` but before `pre`. For the syntax to set the database locale, see "DB_LOCALE" on page 2-9.

The Server Locale

The *server locale*, which is set with the **SERVER_LOCALE** environment variable, 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
- File, **sqexplain.out** that the SQL statement SET EXPLAIN generates

The database server does not use the server locale, however, 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 using the server locale, see Chapter 4, "Database Server Features."

The database server looks for product-specific message files in the message directory that is associated with the locale specified in **SERVER_LOCALE**. For more information, see "Locating Message Files" on page 1-45.

In the example connection that Figure 1-2 on page 1-23 shows, the Informix database server uses the locale specified in **SERVER_LOCALE** to determine the code set to use when it writes a message-log file. For the syntax to set the server locale, see "SERVER_LOCALE" on page 2-32.

Tip: *The database server is the only IBM 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.*

The server locale and the server-processing locale are two different locales. For more information about the server-processing locale, see "Determining the Server-Processing Locale" on page 1-36.

The Default Locale

IBM Informix products use U.S. English as the *default locale* if you do not set the environment variables that can specify a locale.

The default 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 database applications requires no special steps. To use a customized version of U.S. English, British English, or another language, however, your environment must identify the appropriate locale.

For information on how to specify a GLS locale, see "Setting a Nondefault Locale" on page 1-31.

The Default Code Set

The default locale, U.S. English, has the following locale name, where `en` indicates the English language, `us` indicates the United States territory, and the numbers indicate the platform-specific name of 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 *IBM Informix Guide to SQL: Syntax*. The following table describes the default code set for UNIX and for Windows platforms.

Platform	Default Code Set
UNIX	ISO8859-1
Windows	Microsoft 1252

UNIX

Windows

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 "Code-Set-Conversion Filenames" on page A-12.

Default End-User Formats for Date and Time

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

- For DATE values: `%m/%d/%iy`
- For DATETIME values: `%iY-%m-%d %H:%M:%S`

For information about these formatting directives, see "GL_DATE" on page 2-16 and "GL_DATETIME" on page 2-25. For an introduction to date and time end-user formats, see "Date and Time Formats" on page 1-20. For information about how to customize these end-user formats, see "Customizing Date and Time End-User Formats" on page 1-46.

Default End-User Formats for Numeric and Monetary Values

When you use the default locale, IBM 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 plus (+) and minus (-), respectively.

For monetary values, IBM Informix products also use a currency symbol, the dollar ($) sign, in front of a monetary value. For an introduction to numeric and monetary end-user formats, see "Numeric and Monetary Formats" on page 1-19. For information about how to customize these end-user formats, see "Customizing Monetary Values" on page 1-48.

Setting a Nondefault Locale

By default, IBM Informix products use the U.S. English locale, but IBM Informix products support many other locales.

To use a nondefault locale, you must set the following environment variables:

- Set the **CLIENT_LOCALE** environment variable to specify the appropriate 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 the database locale for a client application to use when it connects to a database.

  If you do not set **DB_LOCALE** on the client system, the client application sets **DB_LOCALE** to the client locale. This default value avoids the need for the client application 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 the appropriate server locale.

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

To access a database that has a nondefault locale, the **CLIENT_LOCALE** and **DB_LOCALE** settings on the client system must support this nondefault locale. Both locales must be the same, or their code sets must be convertible, as described in "Performing Code-Set Conversion" on page 1-40.

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

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 1-33.

Using GLS Locales with IBM Informix Products

IBM 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 IBM Informix products that display product-specific messages look in a directory specific to the client locale to find these messages.

Supporting Non-ASCII Characters

An IBM Informix product obtains its code set from its GLS locale. Locales are available for both single-byte and multibyte code sets. All supported code sets 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 "Code Sets for Character Data" on page 1-11.

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 IBM Informix product is installed. For more information on the default code set, "The Default Code Set" on page 1-29.

- Many nondefault locales support the default code set.

  UNIX

  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 unified Chinese code set is GB18030-2000.)

For the contexts in which you can use non-ASCII characters, including multibyte characters, see Chapter 3, "SQL Features," Chapter 4, "Database Server Features," and Chapter 5, "General SQL API Features."

For an IBM Informix product to support non-ASCII characters, however, it must use a locale that supports a code set with the 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**
    If **CLIENT_LOCALE** is not set, the client sets it to the default locale.
  - **DB_LOCALE**
    If **DB_LOCALE** is not set, the client does not send a **DB_LOCALE** value to the database server.
- User-customized end-user formats
  - Date and time end-user formats: **GL_DATE** and **GL_DATETIME**
  - 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 these settings to extract the following information:

- How are numeric and monetary values formatted?
- How are dates and times formatted?
- What database locale does the client expect?

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

Verifying the Database Locale

To open an existing database, the client application must correctly identify the database locale for that database. To verify the database locale, the database server compares the following two locales:

- The locale specified by **DB_LOCALE** that the client application sends
- 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 locale modifiers are the same. If these database locales do not match, the database server performs the following actions:

- It sets the eighth character field of the SQLWARN array in the SQL Communications Area (SQLCA structure) to `W` as a warning flag. Values for `W` are ASCII 32 (blank) and ASCII 87 (W).
- It uses the database locale that is stored in the system catalog of the requested database as the database locale.

Warning: *Check for the SQLWARN warning flag after your 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.*

Checking for Connection Warnings

To check for the eighth character field of the SQLWARN array, an ESQL/C client application can check the **sqlca.sqlwarn.sqlwarn7** 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.

For more information on how to handle exceptions within an ESQL program, see the *IBM Informix ESQL/C Programmer's Manual*.

Important: *Array elements in SQLWARN arrays are numbered starting with zero in IBM Informix ESQL/C, but starting with one in other languages. For IBM Informix GLS tools that use 1-based counts on arrays, such as IBM Informix 4GL and IBM Informix Dynamic 4GL, the warning character that IBM Informix ESQL/C calls* ***sqlca.sqlwarn.sqlwarn7*** *is called* ***SQLCA.SQLAWARN[8]****.*

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 system and the database.

Once the Informix database server verifies the database locale, it uses a precedence of environment variables from the client and database locales to set the server-processing locale.

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

- Locale information for the database

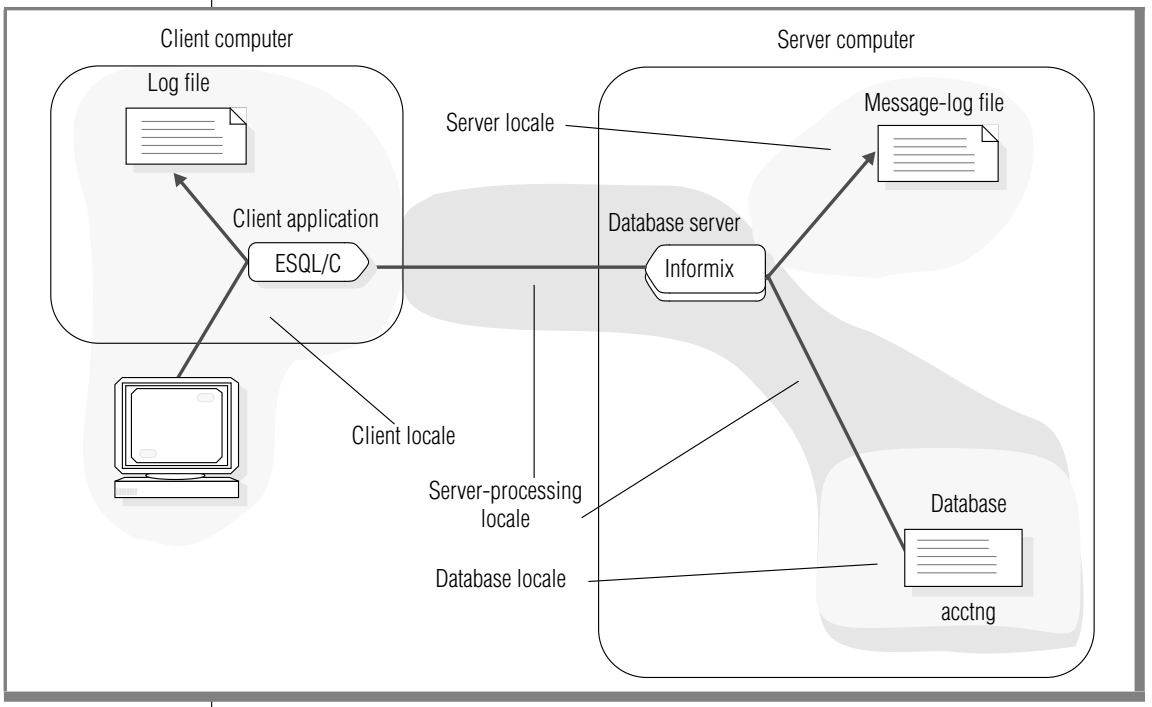
  This database information includes the localized order and code set for data in columns with the locale-sensitive data types (NCHAR and NVARCHAR). The database server obtains this information from the name of the database locale that it has just verified.

- Locale information for client-application data

  This client-application information provides the end-user formats for date, time, numeric, and monetary data. The database server obtains this information from the client application when the client requests a connection.

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



Tip: *The database server uses the server locale, as specified by the* ***SERVER_LOCAL*** *environment variable, for read and write operations on its own operating-system files. For information about operating-system files, see* *"GLS Support by Informix Database Servers" on page 4-4*.

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. 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.
2. The locale that the **DB_LOCALE** environment variable on the server computer indicates
3. The default locale (U.S. English)

IDS

Dynamic Server uses the precedence of steps 1, 2, and 3 in the preceding list to obtain the database information for the server-processing locale. You are not required to set the other environment variables. ♦

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 "Locale Categories" on page A-3.*

For more information on how the database server obtains these environment variables, see "Sending the Client Locale" on page 1-34.

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.
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 locale (**en_us.8859-1**), 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 locale (**fr_fr.8859-1**), the database server must reestablish the server-processing locale. The database server sets the eighth character field of the SQLWARN array to `W` indicate that the two locales are different. The client application, however, 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 locale (**ja_jp.sjis**) 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 "End-User Formats" on page 1-17.

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)

   Support for **DBDATE** and **DBTIME** provides backward compatibility for client applications that are based on earlier versions of IBM Informix products. It is recommended that you use **GL_DATE** and **GL_DATETIME** for new applications.

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

Tip: *The precedence rules apply to how the database server determines the NUMERIC, MONETARY, TIME, and MESSAGES categories of the server-processing locale. For more information on these locale categories, see "Locale Categories" on page A-3.*

The client application passes the **DBDATE, DBMONEY, DBTIME, GL_DATE,** and **GL_DATETIME** 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 1-34.

Performing Code-Set Conversion

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).

IBM Informix products use GLS locales to perform code-set conversion. Both an IBM Informix client application and a database server might perform code-set conversion. For details, see "Database Server Code-Set Conversion" on page 4-5 and "Client Application Code-Set Conversion" on page 5-4.

You specify a code set as part of the GLS locale. At runtime, IBM 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 *database code set*, which the **DB_LOCALE** environment variable specifies, to transfer data to and from the database.
- The database server uses the *server code set*, which the **SERVER_LOCALE** environment variable specifies, to write files (such as debug and warning files).

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 written, 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 cannot 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, 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 Code-Set Conversion Is Performed

An application must use code-set conversion only if 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 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.

Tip: *IBM Informix products support IBM CCSID code-set numbers, a system of 16-bit numbers that uniquely identify the coded graphic character representations. For more information, see* *Appendix A, "Managing GLS Files."*

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

  For example, the code sets **ccdc** and **big5** are both internal representations of a subset of the Chinese language. These subsets, however, include different numbers of Chinese characters.

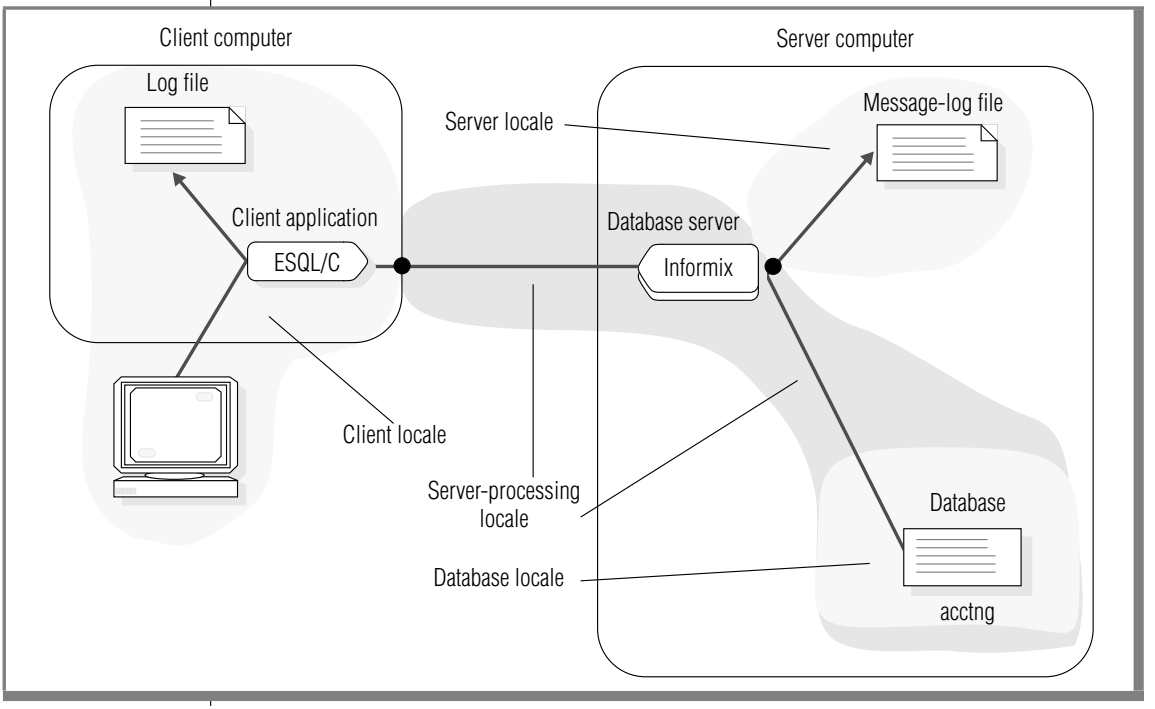
Important: *GLS fully supports the unified Chinese GB18030-2000 code set, including all characters in the Unicode Basic Multilingual Plane (BMP) and in the extended planes.*

If a code-set conversion is required for data transfer from computer A to computer B, then it is also required for data transfer 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. For more information, see "Client Application Code-Set Conversion" on page 5-4.
- 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. For more information, see "Database Server Code-Set Conversion" on page 4-5.

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 example 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.

UNIX

IBM Informix Gateway products on UNIX use the **GL_PATH** environment variable to override the default locations for GLS code-set conversion tables. For details, see the Version 7.2 or later IBM Informix Enterprise Gateway documentation. ♦

Locating Message Files

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

Platform	Directory
UNIX	**$INFORMIXDIR/msg/*lg_tr*/*code_set***
Windows	**%INFORMIXDIR%\msg*lg_tr**code_set***

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-8.

IBM Informix products use a precedence of environment variables to locate product-specific message files. The **DBLANG** environment variable lets you override the client locale for the location of message files that IBM Informix products use. You might use **DBLANG** to specify a directory where the message files reside for each locale that your environment supports.

Customizing 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 an introduction to end-user formats, see "End-User Formats" on page 1-17.

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, 10-27-97 for the date 10/27/97) 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. *(Supported for backward compatibility.)*
DBTIME	Specifies a time end-user format for certain embedded-language (ESQL) library functions. *(Supported for backward compatibility.)*

A date or time end-user format string specifies a format for the manipulation of internal DATE or DATETIME values as a literal string.

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).

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.

Tip: *DBDATE and DBTIME can also provide some support for era-based dates.*

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

- For either **GL_DATE** or **GL_DATETIME**, `E` can appear in several formatting directives.

  For a list of valid formatting conversions for eras, see "Alternative Time Formats" on page 2-27.
- For **DBDATE**, `E` can appear in the format specification.

Date and Time Precedence

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

1. **DBDATE**
2. **GL_DATE**
3. Information defined in the client locale (if **CLIENT_LOCALE** is set)
4. Default date format is `%m/%d/%iy` (if **DBDATE** and **GL_DATE** are not set, and no locale is specified)

IBM 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. Default DATETIME format is `%iY-%m-%d %H:%M:%S` (if **CLIENT_LOCALE**, **DBTIME** and **GL_DATETIME** are not set)

For more information on these formatting directives, see "GL_DATE" on page 2-16 and "GL_DATETIME" on page 2-25.

Customizing Monetary Values

The GLS locales contain end-user formats, which you do not usually need to change. You can set the **DBMONEY** environment variable, however, to customize the appearance of the currency notation. For information on the **DBMONEY** environment variable, see the *IBM 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. IBM 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 is not set, the client locale is the default locale.
3. Default currency notation = $,.

   If **DBMONEY** is not set, and no locale is specified, the currency symbol is the dollar sign, the thousands separator is the comma, and the decimal separator is the period.

Chapter

2

GLS Environment Variables

In This Chapter

IBM 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. For more information about environment variables, see the *IBM Informix Guide to SQL: Reference*.

Setting and Retrieving Environment Variables

The GLS feature lets you use the diacritics, collating sequence, and monetary, date, and number conventions of the language that you select when you create databases. No ONCONFIG configuration parameters exist for GLS, but you must set the appropriate environment variables.

E/C

With IBM Informix ESQL/C, you can use the C **putenv( )** function to modify, create, and delete environment variables, and the C **getenv( )** function to retrieve the values of environment variables from the operating-system environment. For details, see the *IBM Informix ESQL/C Programmer's Manual*. ♦

UNIX

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

Windows

On Windows, set environment variables in the **InetLogin** structure or use the **Setnet32** utility to set environment variables in the **registry**. For more information about **InetLogin**, see the Microsoft Windows documentation for your SQL API. For more information about **Setnet32**, see your *Installation Guide*. ♦

Important: *If you use **ifx_putenv( )**, the application must set all environment variables before it calls any other Informix library routine to avoid initializing the GLS routines and freezing the values of certain locale and formatting environment variables.*

GLS-Related Environment Variables

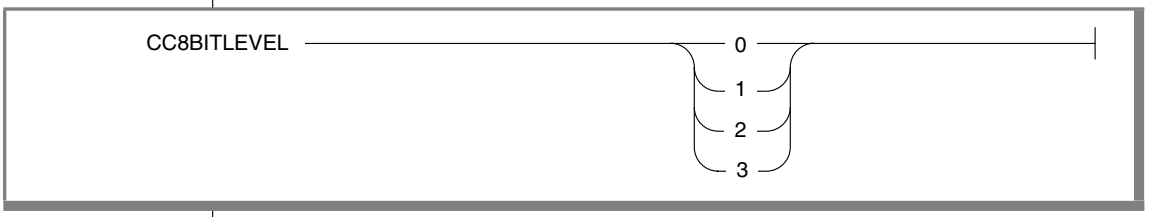
This section lists the GLS-related environment variables that you can set for Informix database servers and SQL API products.

UNIX

IBM Informix Gateway products on UNIX use the **GL_PATH** environment variable to override the default locations for GLS code-set conversion tables. For details, see the Enterprise Gateway documentation. ♦

CC8BITLEVEL

The **CC8BITLEVEL** environment variable determines the type of processing that the ESQL/C filter, **esqlmf**, performs on non-ASCII (8-bit and multibyte) characters. See also "Generating Non-ASCII Filenames" on page 6-6.



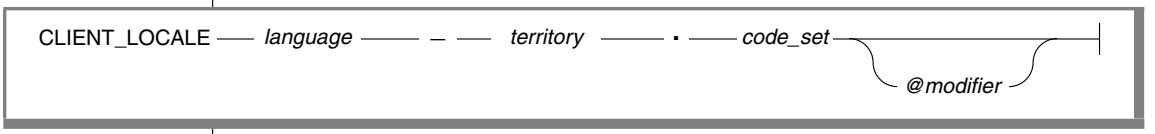
Element	Description
0	The **esqlmf** filter converts all non-ASCII characters in literal strings and comments to octal constants (for C compilers that do not support these uses of non-ASCII characters).
1	The **esqlmf** filter converts non-ASCII characters in literal strings to octal constants but allows them in comments (some C compilers do support non-ASCII characters in comments).
2	The **esqlmf** filter allows non-ASCII characters in literal strings and ensures that all the bytes in the non-ASCII characters have the eighth bit set (for C compilers with this requirement).
3	The **esqlmf** filter does not 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. 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.*

CLIENT_LOCALE

The **CLIENT_LOCALE** environment variable specifies the *client locale*, which the client application uses in read and write operations, end-user formats, and processing ESQL statements. See also "The Client Locale" on page 1-23.



Element	Description
code_set	Name of the code set that the locale supports
language	Two-character name that represents the language for a specific locale
modifier	Optional locale modifier that has a maximum of four alphanumeric characters.
territory	Two-character name that represents the cultural conventions. For example, *territory* might specify the Swiss version of the French, German, or Italian language.

The *modifier* specification modifies the cultural-convention settings that the *language* and *territory* settings imply. The *modifier* usually indicates a special localized collating order that the locale supports.

An example nondefault client locale for a French-Canadian locale follows:

```
CLIENT_LOCALE 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-16. ♦

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

Windows

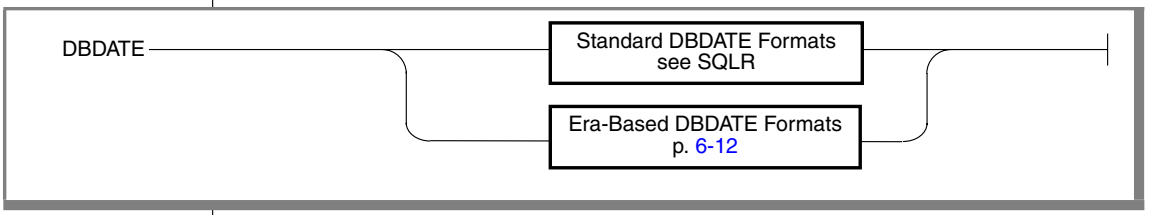
Changes to **CLIENT_LOCALE** also enter in the Windows **registry** database under HKEY_LOCAL_MACHINE. ♦

DBDATE

IBM Informix products support the **DBDATE** environment variable for compatibility with earlier products. We recommend that you use the **GL_DATE** environment variable for new applications.

The **DBDATE** environment variable specifies the end-user formats of values in DATE columns. For information about end-user formats, see "End-User Formats" on page 1-17.

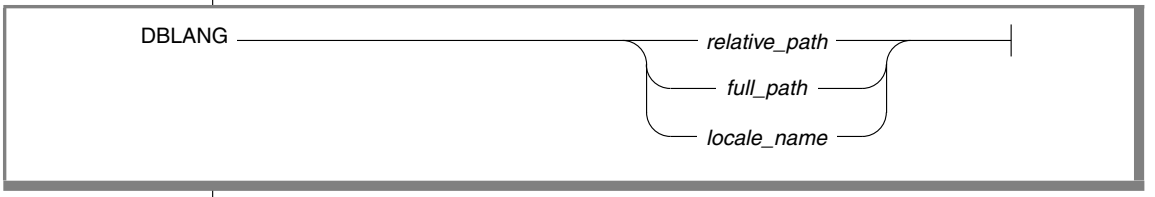
Important: *DBDATE is evaluated at system initialization time. If it is invalid, the system initialization fails.*



Important: *DBDATE variable takes precedence over the **GL_DATE** environment variable, as well as over the default DATE formats that **CLIENT_LOCALE** specifies.*

DBLANG

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



Element	Description
relative_path	Subdirectory of the Informix installation directory (which **INFORMIXDIR** specifies)
full_path	Full pathname of the directory that contains the compiled message files
locale_name	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

IBM 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** on UNIX or **%INFORMIXDIR%\msg\%DBLANG%** on Windows
   b. In **$INFORMIXDIR/$DBLANG** on UNIX or **%INFORMIXDIR%\%DBLANG%** on Windows
3. If **DBLANG** is set to a ***locale_name***: the **msg** subdirectory for the locale in **$INFORMIXDIR/msg/*lg_tr*/*code_set*** on UNIX systems or **%INFORMIXDIR%\msg*lg_tr**code_set*** on Windows, where ***lg***, ***tr***, and ***code_set*** are the language, territory, and code set, respectively, in ***locale_name***.

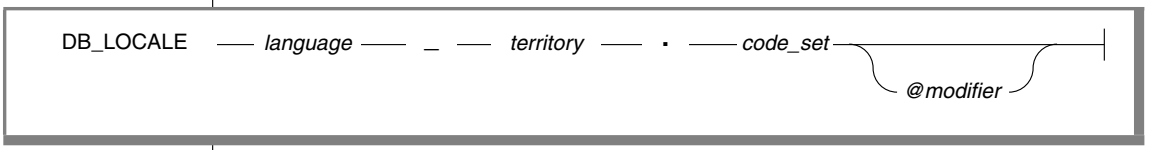
   The value of **DBLANG** does not affect the messages that the database server writes to its message log. The database server obtains the locale for these messages from the **SERVER_LOCALE** environment variable.

4. If **DBLANG** is *not* set: the **msg** subdirectory for the locale in **$INFORMIXDIR/msg/*lg_tr*/*code_set*** on UNIX systems or **%INFORMIXDIR%\msg*lg_tr**code_set*** on Windows, where ***lg*** and ***tr*** are the language and territory, respectively, from the locale that is associated with the IBM Informix product, and ***code_set*** is the condensed name of the code set that the locale supports:
   - For IBM Informix client products: ***lg*** and ***tr*** are from the client locale (from **CLIENT_LOCALE**, if it is set)
   - For Informix database server products: ***lg*** and ***tr*** are from the server locale (from **SERVER_LOCALE**, if it is set)
5. If **DBLANG**, **CLIENT_LOCALE**, and **LANG** are not set:
   a. In **$INFORMIXDIR/msg/en_us/0333** on UNIX systems or **%INFORMIXDIR%\msg\en_us\0333** on Windows, an internal message directory for the default locale
   b. In **$INFORMIXDIR/msg** on UNIX systems or **%INFORMIXDIR%\msg** on Windows, the default Informix message directories

The compiled message files have the **.iem** file extension.

DB_LOCALE

The **DB_LOCALE** environment variable specifies the *database locale*, which the database server uses to process locale-sensitive data. See "The Database Locale" on page 1-26 and Appendix A, "Managing GLS Files.".



Element	Description
code_set	Name of the code set that the locale supports
language	Two-character name that represents the language for a specific locale
modifier	Optional locale modifier that has a maximum of four alphanumeric characters.
territory	Two-character name that represents the cultural conventions. For example, *territory* might specify the Swiss version of the French, German, or Italian language.

The *modifier* specification modifies the cultural-convention settings that the *language* and *territory* settings imply. The *modifier* can indicate a localized collating order that the locale supports. For example, you can set **@***modifier* to specify dictionary or telephone-book collation order.

An example nondefault database locale for a French-Canadian locale follows:

```
DB_LOCALE fr_ca.8859-1
```

UNIX

The **glfiles** utility can generate a list of the GLS locales available on your UNIX system. For more information, see "The glfiles Utility" on page A-16. ♦

IDS

The SET COLLATION statement can specify for the current session a localized collation different from the COLLATION setting of the **DB_LOCALE** locale. This can affect sorting operations on NCHAR and NVARCHAR data values. ♦

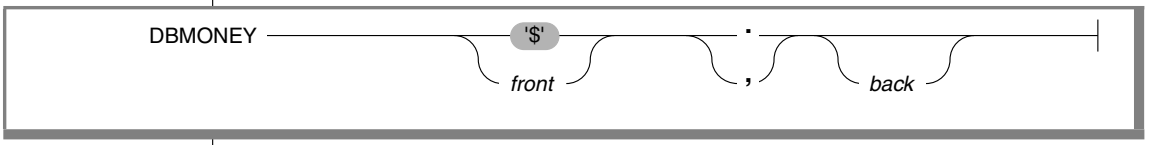
If you do not set **DB_LOCALE** on the client computer, client applications assume that the database locale has the value of the **CLIENT_LOCALE** environment variable. The client application, however, does not send this default value to the database server when it requests a connection.

Windows

Changes to **DB_LOCALE** also enter in the Windows **registry** database under HKEY_LOCAL_MACHINE. ♦

DBMONEY

The **DBMONEY** environment variable specifies the end-user formats for values in MONEY columns. See also "End-User Formats" on page 1-17.



Element	Description
front	Specifies a currency symbol that is displayed before the monetary value.
back	Specifies a currency symbol that is displayed after the value.
, (comma), . (period)	Monetary decimal separator. When you specify the comma or the period, you implicitly assign the other symbol to the thousands separator.

With this environment variable, you can specify the currency notation:

- The currency symbol that appears before or after the monetary value
- The monetary decimal separator, which separates the integral part of the monetary value from the fractional part.

For example, suppose that you use `'DM,'` as the **DBMONEY** seetting. This **DBMONEY** setting specifies the following currency notation:

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

The *front* or *back* symbol can be non-ASCII character if your client locale supports a code set that defines the non-ASCII character. Any symbol that is not a letter must be enclosed within quotation marks. Period or comma are not valid *front* or *back* symbols. In the default locale, the dollar ($) sign is the default *front* currency symbol, period (.) is the default decimal separator, and comma (,) is the default thousands separator. The **DBMONEY** setting takes precedence over any notation defined by the MONETARY category of the locale. See also "Customizing Monetary Values" on page 1-48.

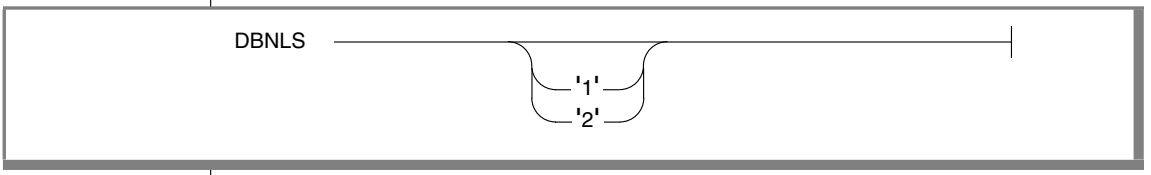
Most GLS locales for European languages can support codesets that include the euro symbol for monetary values.

IDS

DBNLS

The **DBNLS** environment variable specifies whether automatic data type conversion is supported between NCHAR and NVARCHAR database columns and CHAR and VARCHAR variables (respectively) of the client systems.

Global Language Support (GLS) does not require the **DBNLS** environment variable. But Dynamic Server databases continue to support the legacy behavior of **DBNLS**, which supports applications that manipulate tables with NCHAR or NVARCHAR columns.



For UNIX systems that use the C shell, you can enable client applications such as DB-Access, IBM Informix SQL, IBM Informix 4GL, IBM Informix Dynamic 4GL, and embedded-SQL applications such as ESQL/C to convert automatically between CHAR and VARCHAR variables of the client application and NCHAR and NVARCHAR columns of the database by this command:

```
setenv DBNLS 1
```

This setting also supports the automatic conversion of values retrieved from NCHAR columns into CHAR variables, and the conversion of NVARCHAR column values into VARCHAR variables.

Similarly, when **DBNLS** = 1, character strings stored as CHAR variables can be inserted into NCHAR columns, and character strings stored as VARCHAR variables can be inserted into NVARCHAR database columns.

To support these features, **DBNLS** must also be set to 1 on the client system. This setting also enables the client system to display dates, numbers, and currency values in formats specified on the client locale.

Conversely, setting no value for **DBNLS** disables automatic conversion between CHAR and VARCHAR variables of the client application and NCHAR and NVARCHAR columns of the database, and also prevents Dynamic Server from using the locale files of the client system:

```
setenv DBNLS
unsetenv DBNLS
```

Another possible setting for **DBNLS** is 2, by using this command:

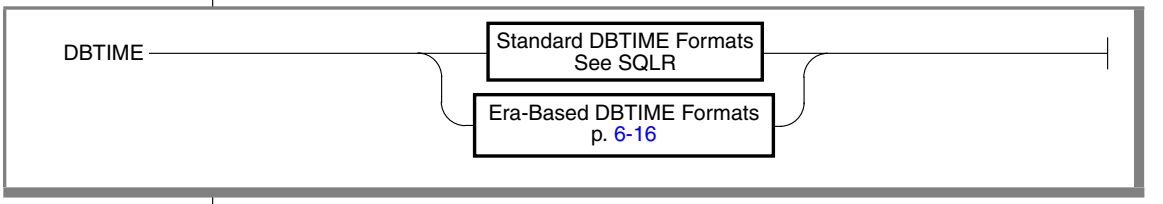
```
setenv DBNLS 2
```

This supports automatic data type conversion between NCHAR and CHAR and between NVARCHAR and VARCHAR (if the client system has **DBNLS** set to 1 or 2). The client and the database server can have different locales.

E/C

DBTIME

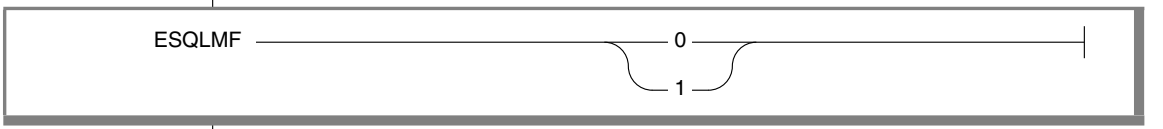
IBM Informix products support **DBTIME** for compatibility with earlier products. We recommend that you use the **GL_DATETIME** environment variable for new applications. The **DBTIME** environment variable specifies the end-user formats of values in DATETIME columns for SQL API routines. See also "End-User Formats" on page 1-17.



Tip: *DBTIME affects only certain formatting routines in the ESQL/C function libraries. See "DATETIME-Format Functions" on page 6-15.*

ESQLMF

The **ESQLMF** environment variable can have the values 1 or 0.



Element	Description
0	**esql** compiles source code whose non-ASCII characters have already been converted.
1	**esql** invokes **esqlmf** to filter multibyte characters when preprocessing an ESQL/C source file.

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

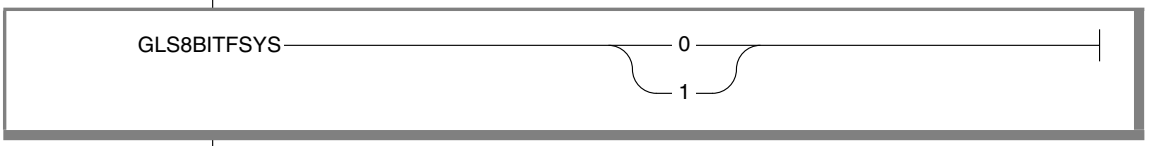
The value of the **CC8BITLEVEL** environment variable determines the type of filtering that **esqlmf** performs. For information about **esqlmf**, see "Generating Non-ASCII Filenames" on page 6-6.

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

Use the **GLS8BITFSYS** environment variable to tell IBM Informix products (such as the ESQL/C processor) whether the operating system is 8-bit clean. This setting determines whether an IBM Informix product can use non-ASCII characters in the filename of an operating-system file that it generates.



Element	Description
0	IBM Informix products assume that the operating tem is not 8-bit clean and generate filenames with 7-bit ASCII characters only.
1	IBM Informix products assume that the operating system is 8-bit clean and can use non-ASCII characters (8-bit or multibyte characters) in the filename of an operating-system file that it generates.

If you include non-ASCII characters in a filename that you specify 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.

For example, create a database that is called **A1A2B1B2**, where **A1A2** and **B1B2** are multibyte characters, with the following SQL statement:

```
CREATE DATABASE A1A2B1B2
```

If **GLS8BITFSYS** is 1 (or is not set) on the server computer, the database server assumes that the operating system is 8-bit clean, and it generates a database directory, **A1A2B1B2.dbs**.

If **GLS8BITFSYS** is set to 0 on the server computer and you include non-ASCII characters in the filename, the IBM 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.

Only some database utilities, such as **dbexport**, and the compilers for IBM Informix ESQL/C products use **GLS8BITFSYS** on the client computer to create and use files. For example, suppose you compile an ESQL/C source file that is called **A1A2B1B2.ec**, where **A1A2** and **B1B2** are multibyte characters. If **GLS8BITFSYS** is set to 1 (or is not set) on the client computer, the ESQL/C processor generates an intermediate C file that is called **A1A2B1B2.c**. For a list of ESQL/C files that check **GLS8BITFSYS**, see "Handling Non-ASCII Characters" on page 6-3.

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 IBM Informix products generate. Before you or an IBM Informix product creates a file and assigns a filename, consider the following questions:

- Does your operating system support non-ASCII filenames?
- Does the IBM 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 of the value of the eighth bit.

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 IBM Informix products use and generate.

Making Sure That Your Product Supports the Same Code Set

Once an IBM Informix product has generated an operating-system file whose filename has non-ASCII characters, it has written that filename and the file contents in a particular code set. Whenever an IBM 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.

The Server Code Set

When the database server creates a file whose filename contains non-ASCII characters, the server locale must support these characters. Before you start a database 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 message log with the UNIX path **/A1A2B1B2/C1C2D1D2**, where **A1A2**, **B1B2**, **C1C2**, and **D1D2** are multibyte characters in the Japanese SJIS code set. To enable the database server to create this message-log file on its computer:

1. Modify the MSGPATH parameter in the ONCONFIG file.

   For UNIX:

   ```
   MSGPATH /A1A2B1B2/C1C2D1D2
   # multibyte message-log filename
   ```

   For Windows:

   ```
   MSGPATH \A1A2B1B2\C1C2D1D2
   # multibyte message-log filename
   ```

2. Set the **SERVER_LOCALE** environment variable on the server computer to the Japanese SJIS locale, **ja_jp.sjis**.
3. Start the database server with the **oninit** utility.

When the database server initializes, it assumes that the operating system is 8-bit clean and creates the **/A1A2B1B2/C1C2D1D2** message log on UNIX, or the **\A1A2B1B2\C1C2D1D2** file on Windows.

The Client Code Set

When an ESQL/C processor 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 **/A1A2B1B2/C1C2D1D2**, where **A1A2**, **B1B2**, **C1C2**, and **D1D2** 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.

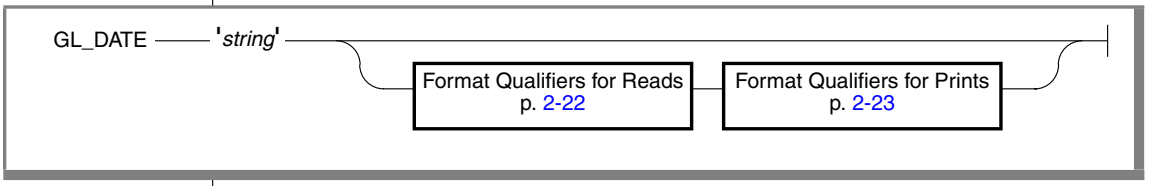
If the code sets that are associated with the filename and with the client locale do not match, a valid filename might contain illegal characters with respect to the client locale. The ESQL/C processor rejects any filename that contains illegal characters and displays the following error message:

```
Illegal characters in filename.
```

GL_DATE

The **GL_DATE** environment variable specifies end-user formats of values for DATE columns. For information about end-user formats, see "End-User Formats" on page 1-17.

Important: *GL_DATE is evaluated when it is used, rather than when it is set. If it is invalid, the operation that called it fails.*



Element	Description
string	Formatting directives that specify the end-user format for **GL_DATE** values. You can use any formatting directive that formats dates.

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

- One or more whitespace characters, which the CTYPE category of the locale specifies
- 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.

The next list describes the formatting directives that are not based on era.

Formatting Directives	Description
%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 an integer (00 through 99).
%d	Is replaced by the day of the month as an integer (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 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 number (00 through 99) for both reading and printing. It is the Informix-specific formatting directive for **%y**.
%iY	Is replaced by the year as a 4-digit number (0000 through 9999) for both reading and printing. It is the Informix-specific formatting directive for **%Y**.
%m	Is replaced by the month as a number (01 through 12).
%n	Is replaced by a newline character.

(1 of 2)

Formatting Directives	Description
%t	Is replaced by the TAB character.
%w	Is replaced by the weekday as a 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 number (00 through 99) for both reading and printing.
%Y	Requires that the year be a 4-digit number (0000 through 9999) for both reading and printing.
%%	Is replaced by % (to allow % in the format string).

(2 of 2)

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 03/05/1997 in the ASCII string format that the following example shows:

```
Mar 05, 1997 (Wednesday)
```

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 IBM Informix product when it tries to format is undefined.

Important: *The setting of the **DBDATE** variable takes precedence over that of the **GL_DATE** environment variable, as well as over the default DATE formats that **CLIENT_LOCALE** specifies.*

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.

For information about end-user formats, see "End-User Formats" on page 1-17.

When an IBM Informix product uses an end-user format to *print* an internal date value as a string, the **%iy** and **%iY** directives perform the same task as **%y** and **%Y**, respectively. To print a year with one of these formatting directives, an IBM 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 5, 1997 formats to '97 03 05'.

- 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 5, 1997 formats to '1997 03 05'.

When an IBM Informix product uses an end-user format to *read* 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 IBM Informix product uses them to read date strings.

	Date String to Read	
GL_DATE Format	**'1994 03 06'**	**'94 03 06'**
%y %m %d	*Error*	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	Internal date for 0094 03 06
%iY %m %d	Internal date for 1994 03 06	Internal date for 1994 03 06

In a read of a date string, the **%iy** and **%y** formatting directives both prefix the first two digits of the current year to expand any 1-digit or 2-digit year. You can set the **DBCENTURY** environment variable to change this default.

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.
- `O` (the letter O) indicates use of locale-defined alternative digits.

These date-formatting directives can support conversion modifiers.

Date Format	Description
%EC	Accepts either the full or the abbreviated era name for reading; 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 the full or the abbreviated era name for reading. 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** (or **%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).

(1 of 2)

Date Format	Description
%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** (or **%Y** 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 representation for the era (which the **%Ex** directive uses)

The NUMERIC category of the locale defines the alternative digits for a locale (which the **%O***x* formatting directives use).

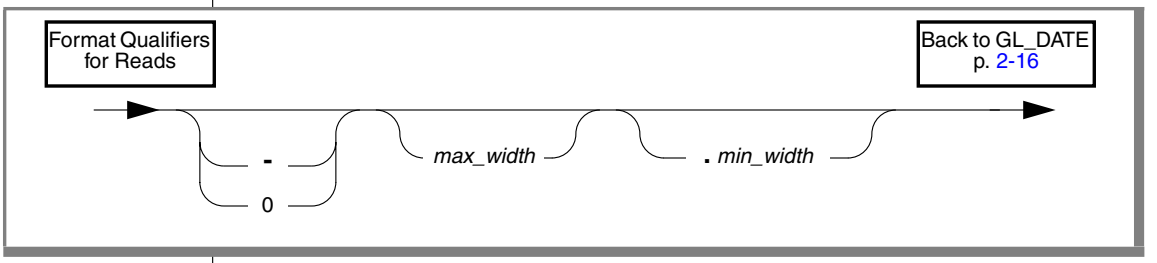
Optional Date Format Qualifiers

You can specify optional *format qualifiers* immediately after the % symbol of the formatting directive. A date format qualifier defines a field specification for the date in read or print operations. The following sections describe what a field specification means for the read and print operations. For information about end-user formats, see "End-User Formats" on page 1-17.

Tip: *The GL_DATETIME environment variable accepts these date format qualifiers in addition to those that "Optional Time Format Qualifiers" on page 2-29 lists.*

Field Specification for Reading a DATE Value

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



Element	Description
- (minus sign)	Field value is *left justified* and begins with a digit; this value can include trailing spaces
0 (zero)	Field value is *right justified*; any zeros on the left are pad characters that are not significant.
max_width	Integer that indicates the maximum number of characters to read
min_width	Integer that indicates the minimum number of characters to read

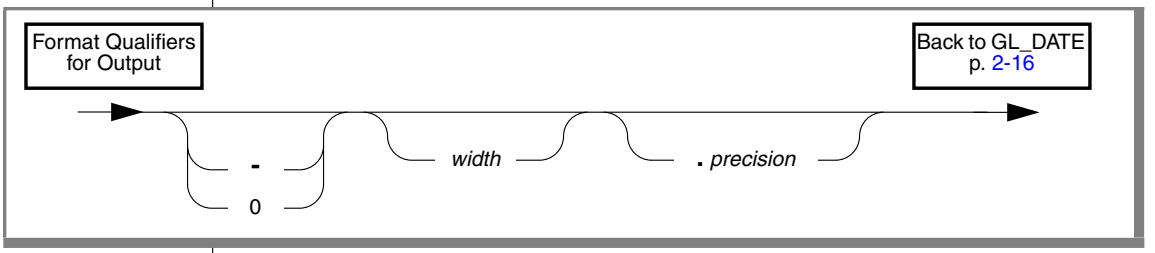
The first character of the field specification indicates whether to assume that the field value is justified or padded. If the first character is neither a minus sign nor a zero, the IBM Informix product assumes that the field value is *right justified* and any spaces on the left are pad characters.

If the field value begins with a zero, however, it cannot include pad characters.

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

Field Specification for Displaying a DATE Value

When an IBM Informix product uses an end-user format to print a date string, the field specification defines the number of characters to print as output. The syntax for the field specification is as follows.



Element	Description
- (minus sign)	Field value is *left justified* and begins with a digit; value can include trailing blank spaces
0 (zero)	Field value is *right justified*; any zeros on the left are pad characters; they are not significant.
width	Integer that indicates a minimum field width for the printed value
*precisio*n	Integer that indicates the precision to use for the field value

The meaning of the *precision* value depends on the 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 *precision* specifies the minimum number of digits to print. If a value supplies fewer digits than *precision* specifies, an IBM 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 *precision* specifies the maximum number of characters to print. If a value supplies more characters than *precision* specifies, an IBM Informix product truncates the value.
%D	Values of *width* and *precision* affect each element of these formatting directives. For example, the field specification %6.4D causes a DATE value to be displayed as if the format were: `%6.4m/%6.4d/%6.4y` where no fewer than four (but no more than six) characters represented the month, day, and year values, in that order, with "/" as the separator.
%O*x*	For formatting directives that include the **O** modifier (alternative digits), the value of *precision* is still the minimum number of digits to print. The *width* value defines the format width rather than the actual number of digits.
%Ex, %EY, %n, %t, %x, %%	Values of *width* and *precision* have no effect on these formatting directives.

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

```
%4m
```

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

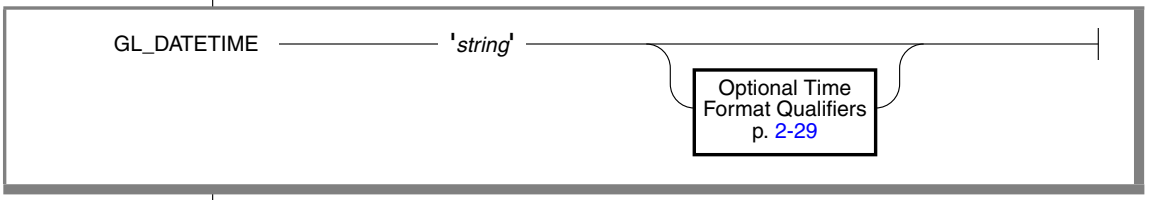
```
%4.3d
```

GL_DATETIME

The **GL_DATETIME** environment variable specifies the end-user formats of values in DATETIME columns. For information about end-user formats, see "End-User Formats" on page 1-17.

A **GL_DATETIME** format can contain the following characters:

- One or more white-space characters, which the CTYPE category of the locale specifies
- 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



Element	Description
string	Contains 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 "GL_DATE" on page 2-16.

This list describes the time formatting directives that are not based on era.

Formatting Directives	Description
%c	Is replaced by a special datetime representation that the locale defines.
%F*n*	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 £ *n* £ 5. This value overrides any width or precision between the % and F character. For more information, see "Optional Time Format Qualifiers" on page 2-29.
%H	Is replaced by the hour as an integer (00 through 23) (24-hour clock).
%I	Is replaced by the hour as an integer (00 through 12) (12-hour clock).
%M	Is replaced by the minute as an integer (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 an integer (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.
%%	Is replaced by % (to allow % in the format string).

White-space or other nonalphanumeric characters must appear between any two formatting directives. Any other characters in the **GL_DATETIME** setting that were not listed above as formatting directives are interpreted as literal characters. If a **GL_DATETIME** format does not correspond to any of the valid formatting directives, the behavior of the IBM Informix product when it tries to format is undefined.

In addition to the formatting directives that the preceding table lists, 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 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, 1997 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/C library functions if the DBTIME environment variable is not set. For information about how these library functions are affected, see* *"DATETIME-Format Functions" on page 6-15**. The value of DBTIME takes precedence over the value 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. It is the same as **%c** if the locale does not define an era.
%EX	Is replaced by a special time representation for the era that the locale defines. It is the same as **%X** if the locale does not define an era.
%OH	Is replaced by the hour in the alternative digits that the locale defines (24-hour clock). It is the same as **%H** if the locale does not define alternative digits).
%OI	Is replaced by the hour in the alternative digits that the locale defines (12-hour clock). It is the same as **%I** if the locale does not define alternative digits).
%OM	Is replaced by the minute with the alternative digits that the locale defines. It is the same as **%M** if the locale does not define alternative digits.
%OS	Is replaced by the second with the alternative digits that the locale defines. It is the same as **%S** if the locale does not define alternative digits.

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

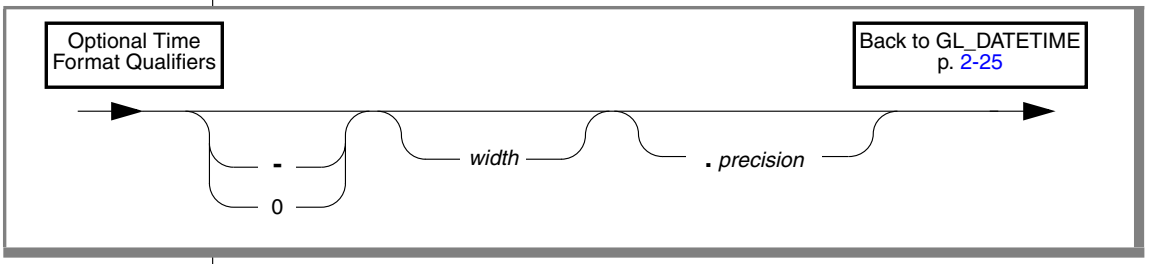
- The full and abbreviated names for an era
- A special date representation for the era (which the **%Ex** formatting directive uses)
- A special time representation for the era (which the **%EX** formatting directive uses)
- A special date/time representation for the era (which the **%Ec** formatting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

The NUMERIC category of the locale defines the alternative digits for a locale (which the **%O*x*** formatting directives use).

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 IBM Informix product reads 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 read operation, see "Field Specification for Reading a DATE Value" on page 2-22. For information about end-user formats, see "End-User Formats" on page 1-17.

When an IBM 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.



Element	Description
- (minus sign)	IBM Informix product prints the field value as *left justified* and pads this value with spaces on the right.
0 (zero)	IBM Informix product prints the field value as *right justified* and pads this value with zeros on the left.
width	Integer that indicates a minimum field width for the printed value
precision	Integer that indicates the precision to use for the field value

The first character of the field specification indicates whether to justify or pad the field value. If the first character is neither a minus sign nor a zero (0), an IBM Informix product prints the field value as *right justified* and pads this value with spaces on the left.

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 *precision* specifies the minimum number of digits to print. If a value supplies fewer digits than the *precision* specifies, an IBM Informix product pads the value with leading zeros. The **%H**, **%M**, and **%S** formatting directives have a default precision of `2`.
%p	Value of *precision* specifies the maximum number of characters to print. If a value supplies more characters than the *precision* specifies, an IBM Informix product truncates the value.
%R, %T	Values of *width* and *precision* affect each element of these formatting directives. For example, the field specification `%6.4R` causes a DATETIME value to be displayed a if the format were this: `%6.4H:%6.4M` Here no fewer than four (but no more than six) characters represented the hour and the minute.
%F	Value of *precision* can follow this directive as an optional precision specification. This value must be between `1` and `5`. Otherwise, an IBM Informix product generates an error. This precision value overrides any *precision* value that you specify between the % symbol and the formatting directive.
%O*x*	For formatting directives that include the **O** modifier, value of *precision* is still the minimum number of digits to print. The *width* value defines the format width, rather than the actual number of digits.
%c, %Ec, %EX, %X	Values of *width* and *precision* have no effect on these formatting directives.

For example, the following formatting directive displays the minute as an integer with a maximum field width of `4`:

```
%4M
```

The following formatting directive displays the hour as an integer with a minimum field width of `3` and a maximum field width of `6`:

```
%6.3H
```

The specified format is applied to all displayed DATETIME values, regardless of their declared precision. For example, suppose that the setting of **GL_DATETIME is** '%Y/%m/%d %H:%M:%S' This setting would cause a value from a DATETIME YEAR TO SECOND column to be displayed as follows:

```
[2000/08/28 14:43:17]
```

If a program executed on August 28 of the year 2000, the same **GL_DATETIME** setting would also display a value from a DATETIME HOUR TO SECOND column as follows:

```
[2000/08/28 14:43:17]
```

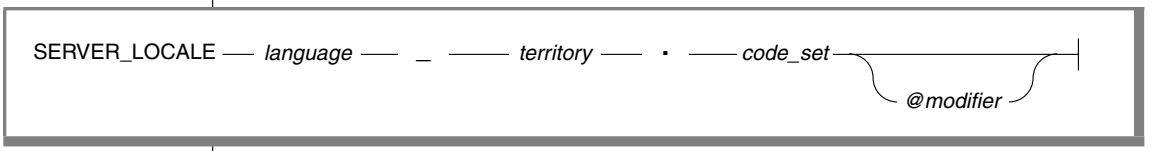
When **GL_DATETIME** is set, every DATETIME value is displayed in the specified format, even if that format includes time units that were not included in the DATETIME qualifier when the data type was declared. Time units outside the declared precision are obtained from the system clock-calendar. To avoid unexpected results, you might prefer to set **GL_DATETIME** only for applications where the DATETIME data types that you display have the same precision as the **GL_DATETIME** setting.

Creation-Time Settings

Like **DBCENTURY**, **DBDATE**, and **GL_DATE**, the **GL_DATETIME** variable can affect how expressions that include literal time values are evaluated. For some earlier releases, resetting environment variables can produce inconsistent behavior in check constraints, triggers, fragmentation expressions, UDRs, and other database objects whose definitions include time expressions. Objects created in this release use the environment variable settings that were in effect at the time when the object was created, rather than the settings at the time of execution (if these are different) to avoid inconsistency.

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 more information about the server locale, see "The Server Locale" on page 1-28 and "GLS Support by Informix Database Servers" on page 4-4.



Element	Description
code_set	Name of the code set that the locale supports
language	Two-character name that represents the language for a specific locale
modifier	Optional locale modifier that has a maximum of four alphanumeric characters.
territory	Two-character name that represents the cultural conventions. For example, *territory* might specify the Swiss version of the French, German, or Italian language.

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

An example nondefault server locale for a French-Canadian locale follows:

```
SERVER_LOCALE 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-16. ♦

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

Windows

Changes to **SERVER_LOCALE** also enter in the Windows **registry** database under HKEY_LOCAL_MACHINE. ♦

Chapter

3

SQL Features

In This Chapter

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

- "Naming Database Objects"
- "Using Character Data Types"
- "Handling Character Data"
- "Using Locale-Sensitive Data Types"
- "Using Data Manipulation Statements"

For more information about the Informix implementation of SQL, see the *IBM Informix Guide to SQL: Syntax*, the *IBM Informix Guide to SQL: Reference*, the *IBM Informix Guide to SQL: Tutorial*, and the *IBM Informix Database Design and Implementation Guide*.

Naming Database Objects

You need to declare names for new database objects (and in some cases. for storage objects, such as dbspaces) when you use data definition language (DDL) statements such as CREATE TABLE , CREATE INDEX, and RENAME COLUMN. This section describes considerations for declaring names for database objects in 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 IBM Informix products. For more information, see* *"Setting a Nondefault Locale" on page 1-31*.

Rules for Identifiers

An SQL identifier is a string of letters, digits, and underscores that represents the name of a database object such as a table, column, index, or view.

A non-delimited SQL identifier must begin with a letter or underscore (_) symbol. Trailing characters in the identifier can be any combination of letters, digits, underscores, or (for Dynamic Server only) dollar ($) signs. Delimited identifiers, however, can include any character in the code set of the database locale; see "Delimited Identifiers" on page 3-9 for more information.

Declaring identifiers that are SQL keywords can cause syntactic ambiguity or unexpected results. For additional information, see the Identifier segment in the *IBM Informix Guide to SQL: Syntax*. See also "Non-ASCII Characters in Identifiers" on page 3-5 and "Valid Characters in Identifiers" on page 3-9.

SQL identifiers can occupy up to 128 bytes on Dynamic Server, or up to 18 bytes on Extended Parallel Server. When you declare identifiers, make sure not to exceed the size limit for your database server. For example, the following statement creates a synonym name of 8 multibyte characters:

```
CREATE SYNONYM A1A2A3B1B2C1C2C3D1D2E1E2F1F2G1G2H1H2 FOR A1A2B1B2
```

XPS

The synonym declared 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 identifiers on Extended Parallel Server. The following CREATE SYNONYM statement, however, generates an error because the total number of bytes in this synonym name is 20:

```
CREATE SYNOMYM A1A2A3B1B2B3C1C2C3D1D2D3E1E2F1F2G1G2H1H2 FOR A1A2B1B2
```

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

Non-ASCII Characters in Identifiers

Informix database servers support non-ASCII (wide, 8-bit, and multibyte) characters from the codeset of the database locale in most SQL identifiers, such as the names of columns, connections, constraints, databases, indexes, roles, SPL routines, sequences, synonyms, tables, triggers, and views.

XPS

On Extended Parallel Server, use only single-byte characters in the following identifiers:

- Dbslice
- Logslice
- Coserver
- Cogroup

Use only ASCII alphanumeric 7-bit names for the following identifiers:

- Chunk name
- Filename
- Message-log filename
- Pathname ♦

IDS

On Dynamic Server, you can use non-ASCII characters (8-bit and multibyte characters) when you create or refer to any of these database server names:

- Chunk name
- Message-log filename
- Pathname

The following restrictions affect the ability of the database server to generate filenames that contain non-ASCII characters:

- The database server must know whether the operating system is 8-bit clean.
- The code set specified by the **DB_LOCALE** setting must support these non-ASCII characters. ♦

In a database with a nondefault locale, whose code set supports multibyte (or other non-ASCII) characters, you can use those non-ASCII characters when you declare most SQL identifiers, as listed in Figure 3-1 on page 3-6.

In the following table, the **Type of Identifier** column lists various categories of objects that can have SQL identifiers or operating-system identifiers. The **SQL Segment** column shows the segment that provides the syntax of the identifier in the *IBM Informix Guide to SQL: Syntax*. The **Example Context** column lists an SQL statement that can declares or can reference the identifier.

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

Type of Identifier	SQL Segment	Example Context
Alias	Identifier	SELECT
IDS Cast	Expression	CREATE CAST ♦
Column name	Identifier	CREATE TABLE
Connection name	Quoted String	CONNECT For more information, see "Specifying Quoted Strings" on page 3-20.
Constraint name	Database Object Name	CREATE TABLE
Cursor name	Identifier	DECLARE For more information, see "Handling Non-ASCII Characters" on page 6-3.
Database name	Database Object Name	CREATE DATABASE
IDS Distinct data type name	Identifier, Data Type	CREATE DISTINCT ♦
IDS Filename	None	LOAD ♦
IDS Function name	Database Object Name	CREATE FUNCTION ♦
Host variable	None	FETCH For more information, see "Handling Non-ASCII Characters" on page 6-3.
Index name	Database Object Name	CREATE INDEX
IDS Opaque data type name	Identifier, Data Type	CREATE OPAQUE TYPE ♦
IDS Operator-class name	Database Object Name	CREATE OPCLASS ♦

(1 of 2)

IDS

IDS

IDS

IDS

Type of Identifier	SQL Segment	Example Context
Routine name	Database Object Name	CREATE FUNCTION♦
Routine name	Database Object Name	CREATE PROCEDURE
Role name	Identifier	CREATE ROLE ♦
Row data type	Identifier	CREATE ROW TYPE ♦
Sequence name	Database Object Name	CREATE SEQUENCE ♦
SQL Statement identifier	Identifier	PREPARE) For more information, see "Handling Non-ASCII Characters" on page 6-3.)
SPL routine name	Database Object Name	CREATE PROCEDURE
SPL routine variables	None (language-specific)	CREATE PROCEDURE FROM
Synonym	Database Object Name	CREATE SYNONYM
Table name	Database Object Name	CREATE TABLE
Trigger correlation name	Database Object Name	CREATE TRIGGER
Trigger name	Database Object Name	CREATE TRIGGER
View name	Database Object Name	CREATE VIEW

(2 of 2)

Qualifiers of SQL Identifiers

The **SQL Segment** column in Figure 3-1 on page 3-6 refers to the segment in the *IBM Informix Guide to SQL: Syntax* that describes the syntax of the identifier. In many cases, the complete syntax can include other identifiers. For example, the Database Object Name segment shows that the syntax of an index name can also include a *database* name, a database *server* name, and an *owner* name, as well as the unqualified name of the index.

Keep in mind that even if the simple, unqualified name of a database object accepts multibyte characters, other identifiers in the fully-qualified name of that object, such as *database***@***server***:***owner***.***index*, can include multibyte characters only if they also appear in the previous table. In this example, the *database* qualifier within the fully-qualified index name can include multibyte characters, but the identifier of the database *server* that qualifies the *index* name cannot include multibyte characters.

Owner Names

The *owner name* is the name of the user (or of a pseudo-user, for an *owner* like **informix** that does not correspond to the login name of an actual user) who is associated with the creation of a database object. The owner name qualifies the identifier of the database object, which the owner typically can modify or drop. A synonym for the term *owner name* is *authorization identifier*.

ANSI

The ANSI term for *owner name* is *schema name*. In an ANSI-compliant database, you must specify the *owner* name as a qualifier of the identifier of any database object that you do not own. ♦

Non-ASCII characters are not valid in an owner name unless your operating system supports those characters in user names.

UNIX

If your database server is on a UNIX system, the owner-name qualifier defaults to the UNIX login ID. Most versions of UNIX, however, do not support multibyte characters in UNIX login IDs.

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

In some East Asian locales, an owner name can include multibyte characters when you create database objects and specify an explicit owner. For example, you can assign an owner name that contains multibyte characters when you specify the owner of an index (within single quotes) in a CREATE INDEX statement. The following statement declares an index with 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 A^1A^2, B^1B^2, and C^1C^2 are valid characters in this code set.

Pathnames and Filenames

Valid pathnames and filenames are operating system-dependent; see "Handling Non-ASCII Characters" on page 6-3. Multibyte characters in hard-coded pathnames, for example, limits the portability of your application to operating systems that can support multibyte filenames.

XPS

For Extended Parallel Server, only ASCII alphanumeric 7-bit characters are valid in pathnames or in filenames. ♦

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 strings of characters in double (") quotes as delimited identifiers and strings of characters in single (') quotes as data strings. This interpretation of single- and double-quotes is compliant with the ANSI/ISO standard for SQL.

In a nondefault locale, you can specify valid non-ASCII characters of the current codeset 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 3-6 indicates that you can specify non-ASCII characters in the declaration of an index name. Thus, you can include non-ASCII characters in an undelimited index name, or in an index name that you have enclosed in double quotes to make it a delimited identifier, as in the following SQL statement:

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

For a description of delimited identifiers, see the Identifier segment in the *IBM Informix Guide to SQL: Syntax*.

Valid Characters in Identifiers

In an SQL identifier, a *letter* can be any character in the alpha class that the locale defines. The alpha class lists all characters that are classified as alphabetic. For more information about character classification, see "The CTYPE Category" on page A-4. 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`. SQL identifiers can use these ASCII characters wherever *letter* is valid in an SQL identifier.

In a nondefault locale, the alpha class of the locale also includes the ASCII characters in the ranges a to z and A to Z. It might also include non-ASCII characters, such as letters from non-Roman alphabets (such as Greek or Cyrillic) or ideographic characters. For example, the alpha class of the Japanese UJIS code set (in the Japanese UJIS locale) contains Kanji characters. When IBM Informix products use a nondefault locale, SQL identifiers can use non-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 declares the identifier **marché**, which includes the 8-bit character **é**, for the database.
- The CREATE TABLE statement declares the identifier **équipement**, which includes the 8-bit character **é**, for the table, and the identifiers **code**, **description**, and **prix_courant** for the columns.
- The CREATE VIEW statement declares the identifier **çà_va**, which includes the 8-bit characters **ç** and **à**, for the view.
- The SELECT clause within the CREATE VIEW statement specifies the identifiers **numéro** and **nom** as columns in the projection 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 within a French locale. For these names to be valid, the database locale must support a code set that includes these French characters in its alpha class.

For the syntax and usage of identifiers in SQL statements, see the Identifier segment in the *IBM Informix Guide to SQL: Syntax*.

Using Character Data Types

The locale affects the collation of built-in SQL character data types:

- Character types that use localized collation: NCHAR and NVARCHAR
- Character types that use code-set order for collation:
  - CHAR
  - LVARCHAR ♦
  - VARCHAR
  - TEXT

IDS

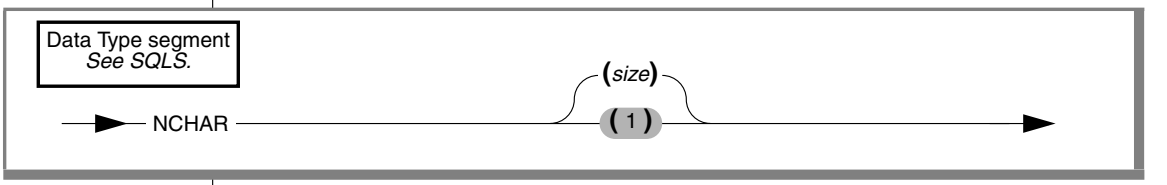
The *IBM Informix Guide to SQL: Reference* describes these types. For information about collation, see "Character Classes of the Code Set" on page 1-13.

Localized Collation of Character Data

The choice of locale can affect the collating order of NCHAR and NVARCHAR character data types, as the following sections describe.

The NCHAR Data Type

The NCHAR data type stores character data in a fixed-length field as a string of single-byte or multibyte letters, numbers, and other characters that are supported by the code set of your database locale. The syntax of the NCHAR data type is as follows.



Element	Purpose
size	Specifies the number of bytes in the column. The total length of an NCHAR column cannot exceed 32,767 bytes. If you do not specify *size*, the default is NCHAR(1).

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

NCHAR is a locale-sensitive data type. The only difference between NCHAR and CHAR data types is the collation order. The database server sorts data in NCHAR columns in localized order, if the locale defines a localized order. (For most operations, the database server collates data in CHAR columns in code-set order, even if the locale defines a localized collation.)

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

Handling NCHAR Data

A client application manipulate NCHAR data using the **CLIENT_LOCALE** setting of the client system. The client application performs code-set conversion of NCHAR data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**, and **DBNLS** is set to `1`.

Multibyte Characters with NCHAR

To store multibyte character data in an NCHAR column, your database locale must support a code set that includes the 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, rather than to the number of logical characters.

Because one multibyte character requires 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 *IBM Informix Guide to SQL: Reference* provides detailed reasons why you should not store certain numberic values in 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.

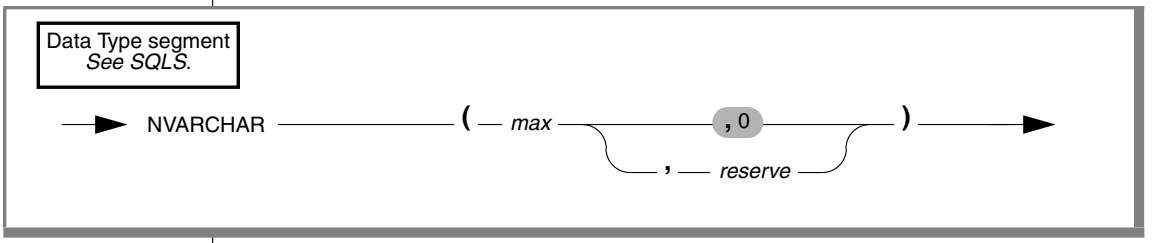
Nonprintable Characters with NCHAR

An NCHAR value can include tabs, spaces, and other whitespace and nonprintable characters. Nonprintable NCHAR and CHAR values are entered, displayed, and treated similarly.

The NVARCHAR Data Type

The NVARCHAR data type stores character data in a variable-length field. Data can be a string of single-byte or multibyte letters, numbers, and other characters that are supported by the code set of your database locale.

The syntax of the NVARCHAR data type is as follows:



Element	Purpose
max	Specifies the maximum number of bytes that can be stored in the column.
reserve	Specifies the minimum number of bytes that can be stored in the column.

You must specify *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.

Specify the *reserve* parameter when you initially intend to insert rows with data values having few or no characters in this column but later expect the data to be updated with longer values. This value can range from 0 to 255 bytes but must be less than the *max* size of the NVARCHAR column. If you do not specify a minimum space value, the default value of *reserve* is 0.

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 whitespace, 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.

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, if the locale defines a localized order. For most operations, the database server collates data in CHAR columns in code-set order.

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 about code-set conversion, see "Performing Code-Set Conversion" on 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 uses several bytes for storage, the value of *max* bytes does not indicate the number of logical 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 the column can store. Make sure to declare the *max* value of the NVARCHAR column so that it can hold enough multibyte 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.

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

Storing Numeric Values in an NVARCHAR Column

The database server does not pad a numeric value in a NVARCHAR column 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 NVARCHAR(10));
INSERT INTO mytab VALUES (1);
```

Performance Considerations

The NCHAR data type is similar to the CHAR data type, and NVARCHAR is similar to the VARCHAR data type. These data types differ in two ways:

- 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 depends 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 access 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, LVARCHAR, and VARCHAR data types.

Other Character Data Types

The choice of locale can affect the following character data types, which are individually described in sections that follow:

- CHAR
- VARCHAR
- LVARCHAR ♦ IDS
- TEXT

The CHAR Data Type

The CHAR data type stores character data in a fixed-length field. Data can be a string of single-byte or multibyte letters, numbers, and other characters that are supported by the code set of your database locale.

This list summarizes how the 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.
- 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**.

The VARCHAR Data Type

The VARCHAR data type stores character strings of up to 255 bytes in a variable-length field. Data can consist of letters, numbers, and symbols. CHARACTER VARYING is handled exactly the same as VARCHAR. 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.
- 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**.

IDS

The LVARCHAR Data Type

The LVARCHAR data type can store character strings of up to 32,739 bytes in a variable-length field. If you specify no *maximum size* in its declaration, the default upper size limit is 2048 bytes. Data values can include letters, numbers, symbols, whitespace, and unprintable characters.

LVARCHAR is similar to the VARCHAR data type in several ways:

- Strings of the LVARCHAR data type are collated in code-set order.
- Client applications perform code-set conversion on LVARCHAR data.
- LVARCHAR supports the built-in SQL length functions. (See "Using SQL Length Functions" on page 3-42.)
- LVARCHAR data type declarations can specify a *maximum size*.

Unlike VARCHAR, however, LVARCHAR has no *reserved size* parameter, and data strings in LVARCHAR columns can be longer than the VARCHAR limit of 255 bytes.

The database server also uses LVARCHAR to represent the external format of opaque data types. In I/O operations of the database server, LVARCHAR data values have no upper limit on their size, apart from file size restrictions or limits of your operating system or hardware resources.

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.
- 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**.

Handling Character Data

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

- Quoted strings
- Comments
- Column substrings
- TRIM function arguments
- UPPER, LOWER, and INITCAP function arguments ♦

IDS

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 string of consecutive characters that is delimited by quotation marks. The quotation marks can be single quotes or double quotes. If the **DELIMIDENT** environment variable is set, however, the database server interprets a string of characters in double quotes as a delimited identifier rather than as a string. For more information about delimited identifiers, see "Non-ASCII Characters in Identifiers" on page 3-5.

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 in the table **mytable**:

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

In this example, the first quoted string includes the multibyte characters A^1A^2 and B^1B^2. The second quoted string includes the multibyte characters X^1X^2 and Y^1Y^2. The third quoted string contains only single-byte characters. This example assumes that the locale supports a multibyte code set with the A^1A^2, B^1B^2, X^1X^2, and Y^1Y^2 characters.

For a description of quoted strings, see the Quoted String segment in the *IBM Informix Guide to SQL: Syntax*.

Specifying Comments

To use comments after SQL statements, introduce the comment text with one of the following comment indicators:

ANSI

+

- The double-hyphen (--) complies with the ANSI SQL standard. ♦
- Braces ({ }) are an Informix extension to the ANSI standard. ♦

In 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.

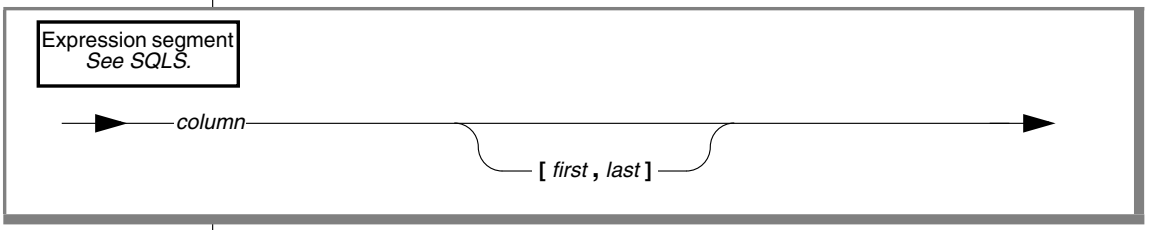
In the following example, the user inserts a column value that includes multibyte characters in the table **mytable**:

```
EXEC SQL insert into mytable
   values ('A1A2B1B2abcd', '123')   -- A1A2 and B1B2 are multibyte
characters.
```

In this example, the SQL comment includes the multibyte characters A^1A^2 and B^1B^2. This example assumes that the locale supports a multibyte code set that includes the A^1A^2 and B^1B^2 characters. For more information on SQL comments and comment indicators, see the *IBM Informix Guide to SQL: Syntax*.

Specifying Column Substrings

In a query (or in any SQL statement containing an embedded SELECT statement), you can use bracket { [ ] } symbols to specify that only a subset of the data in a column of a character data type 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.



Element	Purpose
column	Identifier of a column within a database table or view
first. *last*	Positions of the first and the last byte (respectively) of the retrieved substring

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 **lname** column value is `Albertson`, the query returns these results.

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`. Column substrings, however, are byte based, and the query returns the seventh through ninth *bytes* of the name. 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 the number of characters. If a character column **multi_col** contains a string of three 2-byte characters, this 6-byte string can be represented as follows:

$A^1A^2B^1B^2C^1C^2$

Suppose that a query specified this substring from the **multi_col** column:

```
multi_col[1,2]
```

The query returns the following result:

A^1A^2

The returned substring consists of 2 bytes (1 character), not 2 characters.

To retrieve the first two characters from the **multi_col** column, specify a substring in which *first* is the position of the first byte in the first character and *last* is the position of the last byte in the second character. For the 6-byte string $A^1A^2B^1B^2C^1C^2$, this g expression specifies the substring in your query:

```
multi_col[1,4]
```

The following result is returned:

$A^1A^2B^1B^2$

The substring that the query returns consists of the first 4 bytes of the column value, representing the first two logical 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 $A^1A^2B^1B^2C^1C^2$:

```
multi_col[2,5]
```

The user requests the following bytes in the query: $A^2B^1B^2C^1$. If the database server returned this column substring to the user, however, the first and third logical 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 whitespace characters.

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] = 'A8A2B1B2'
```

The query returns no rows because the database server converts the substring **multi_col[2,4]**, namely the string $A^2A^3A^4$, to three single-byte blank spaces (`sss`). The WHERE clause specifies this search condition:

```
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 blank spaces and creates the following WHERE clause condition:

```
WHERE 'sB¹B²sD¹D²' = 'A¹A²B¹B²'
```

This condition is also never true, so the query retrieves no matching rows.

Errors Involving Partial Characters

Partial characters violate the relational model if the substrings strings can be processed or presented to users in any way that can prevent the concatenation of the substrings from reconstructing the original logical string.

This can occur when a multibyte character has a substring that is 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 your locale is using 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:

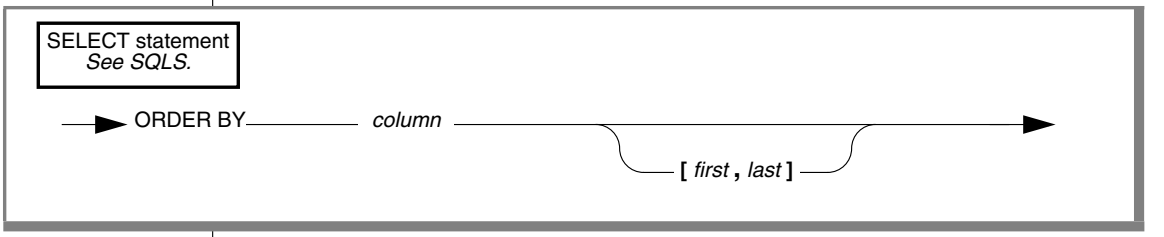
```
WHERE '6' = '6'
```

Partial characters do not occur in single-byte code sets because each character is stored in a single byte. If the database locale supports a single-byte code set, and you specify a column substring in a query, the query returns exactly the requested subset of data; no characters are replaced with whitespace.

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 syntax for specifying column substrings in the ORDER BY clause is as follows.



Element	Purpose
column	Name of a column in the specified table or view.
first. *last*	Positions of the first and last byte (respectively) of the substring

The query results are sorted by the values contained in this column.

XPS

Any column or expression specified in the ORDER BY clause must be listed explicitly or implicitly in the SELECT list of the Projection clause. ♦

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. The more typical scenario, however, 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 wish to retrieve all the rows of the **multi_data** table, and sort the results according to a substring defined as the fourth through sixth characters of the **multi_chars** column, using this query:

```
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, 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.

If the multibyte code set contains a mixture of single-byte characters, 2-byte characters, and 3-byte characters, however, the substring **multi_chars[7,12]** might create partial characters. In this case, you might get unexpected results when you specify a column substring in the ORDER BY clause.

For information on the collation of different types of character data in the ORDER BY clause, see "The ORDER BY Clause" on page 3-30. For the complete syntax and usage of the ORDER BY clause, see the SELECT statement in the *IBM Informix Guide to SQL: Syntax*.

Tip: *A partial character might also be generated when a SQL API copies multibyte data from one buffer to another. For more information, see "Generating Non-ASCII Filenames" on page 6-6.*

Avoidance in TEXT and BYTE Columns

Partial characters are not a problem when you specify a column substring for a column of the TEXT or BYTE data type. The database server avoids partial characters in TEXT and BYTE 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 substring of a BYTE column returns the exact range of bytes that is specified and does not replace any bytes with whitespace characters.

- The database server interprets a TEXT value as a character string.

  A substring from a TEXT column returns the exact range of bytes that is specified. Attempts to resolve partial characters in TEXT data are resource intensive, but the database server does not replace any bytes with whitespace. For more information, see "The TEXT Data Type" on page 3-19.

Warning: *The processing and interpretation of TEXT and BYTE data are the responsibility of the client application, which must handle the possibility of partial characters in these operations.*

Specifying Arguments to the TRIM Function

The TRIM function is a built-in SQL function that removes leading or trailing pad characters from a character string. By default, this pad character is ASCII 32, the blank space. If your locale supports a code set that defines a different whitespace character, TRIM does not remove this locale-specific blank space from the front or back of a string. If you specify the LEADING, TRAILING, or BOTH keywords for TRIM, you can specify a different pad character.

You cannot, however, specify a non-ASCII character as a pad character, even if your locale supports a code set that defines the non-ASCII character.

IDS

LVARCHAR data types are not valid as arguments to the TRIM function. ♦

IDS

Using Case-Insensitive Search Functions

The SQL search functions UPPER, LOWER, and INITCAP support GLS. They accept multibyte characters in character-type source strings and operate on them. The returned data type is the same as the type of the source string:

- UPPER converts every letter in a string to uppercase.
- LOWER converts every letter in a string to lowercase.
- INITCAP changes the first letter of a word or series of words to uppercase.

For complete information about these search functions, see the *IBM Informix Guide to SQL: Syntax*.

Collating Character Data

Collation is the process of sorting 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 "Character Classes of the Code Set" on page 1-13.

By default, the database server sorts strings according to the collation that the **DB_LOCALE** setting implies, and client applications sort according to the **CLIENT_LOCALE** setting, if this is different from the **DB_LOCALE** setting.

IDS

The SET COLLATION statement of Dynamic Server can specify a localized collation different from the **DB_LOCALE** setting for the current session.

See the *IBM Informix Guide to SQL: Syntax* for the syntax of this statement. Database objects that sort strings, such as indexes or triggers, use the collation that was in effect at the time of their creation when they sort NCHAR or NVARCHAR values, if this is different from the **DB_LOCALE** setting. ♦

The collation order of the database server affects SQL statement that perform sorting operations, including CREATE INDEX and SELECT statements.

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:

- The index keys are stored in code-set order when you create an index on columns of these data types:
  - CHAR
  - IDS LVARCHAR ♦
  - VARCHAR

  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 uses 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.

IDS

If the SET COLLATION statement specifies a database locale with localized collation that is different from the **DB_LOCALE** setting, any indexes (and any check constraints) that you subsequently create in the same session always use that localized collation for sorting NCHAR or NVARCHAR strings. ♦

If you use the default locale (U.S. English), the index keys are stored in the code-set order (in ascending or descending order) 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 that uses this locale (or any other locale that does not define a localized collating order) sorts strings from columns of the following data types in code-set order:

- CHAR
- IDS LVARCHAR ♦
- NCHAR
- NVARCHAR
- VARCHAR

Collation Order in SELECT Statements

The SELECT statement performs a queries. Collation order affects the following parts of the SELECT statement:

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

The ORDER BY Clause

The ORDER BY clause sorts 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:

- Columns that are sorted in code-set order:
  - CHAR
  - IDS LVARCHAR ♦
  - VARCHAR
- 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**. This 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 this table shows, names that begin with uppercase letters come before names beginning with lowercase letters, and names that begin with an accented letter (`Ålesund`, `Étaix`, `Ötker`, and `Øverst`) are at the end of the list.

Figure 3-2
Data Set for Code-Set Order of the abonnés 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

(1 of 2)

numéro	nom	prénom
13601	Ålesund	Sverre
13608	Étaix	Émile
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders

(2 of 2)

Results of the query is different, however, 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 the following table 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 abonnés 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	LeMaître	Héloïse
13613	Llanero	Gloria Dolores

(1 of 2)

numéro	nom	prénom
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders
13609	Tiramisù	Paolo Alfredo

(2 of 2)

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 3-24.

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 the only rows are to be retrieved from the **abonnés** table are those 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 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 3-31.

However, if **nom** is an NCHAR column, the database server uses localized order to sort the rows that the WHERE clause specifies. The following example 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 3-32.

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 example of output shows the query results.

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

Because the database server uses the code-set order for the **nom** values, as Figure 3-2 on page 3-31 shows, 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 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
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
13609	Tiramisù	Paolo Alfredo

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. This 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, as Figure 3-2 on page 3-31 shows. 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. 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, as Figure 3-3 on page 3-32 shows, 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 about MATCHES, see the *IBM Informix Guide to SQL: Syntax*.

When a MATCHES expression 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 the same rank in the collation order. For more information about localized order, see "Localized Order" on page 1-15.

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, VARCHAR, or (for Dynamic Server only) LVARCHAR data type, and the string operand of the MATCHES keyword includes brackets ([ ]) to specify a range, sorting follows a localized order, if the locale defines one.

Important: *When the database server determines the characters that fall within a range with the MATCHES operator, it uses the localized order, if* **DB_LOCALE** *or SET COLLATION has specified one, even for CHAR, LVARCHAR, and VARCHAR columns. This behavior is an exception to the rule that the database server uses code-set order for all operations on CHAR, LVARCHAR and VARCHAR columns, and localized order (if one is defined) for sorting operations on NCHAR and NVARCHAR columns.*

Some simple examples show how the database server treats NCHAR, NVARCHAR, LVARCHAR, 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, as Figure 3-3 on page 3-32 shows, 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

(1 of 2)

numéro	nom	prénom
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders

(2 of 2)

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 use only certain symbols as wildcards in the quoted string. For more information about LIKE, see the *IBM Informix Guide to SQL: Syntax*.

The LIKE condition can specify only 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.

The LIKE keyword does not support ranges of characters. That is, you cannot use bracketed characters to specify a range in LIKE conditions.

Wildcard Characters in LIKE and MATCHES Conditions

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

Condition	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 logical characters, not bytes. Therefore, the underscore (_) wildcard of the LIKE clause and the ? (question mark) 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'	`'abA1A2d'`	True
MATCHES	'ab?d'	`'abcd'`	True
MATCHES	'ab?d'	`'abA1A2d'`	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 (\) symbol.

The following MATCHES condition returns a TRUE result for the column value that is shown.

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

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 syntax of these functions, see the Expression segment in the *IBM 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 an SPL routine 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 whitespace character.

This table assumes that all arguments consist of single-byte characters.

LENGTH Argument	Behavior	Example
Quoted string	Returns number of bytes in string, minus any trailing whitespace (as defined in the locale).	If the string is `'Ludwig'`, the result is `6`. If the string is `'Ludwigssss'`, the result is still `6`.
CHAR, VARCHAR, LVARCHAR, NCHAR, or NVARCHAR column	Returns number of bytes in a column, minus any trailing white-space characters, regardless of defined length 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 `6`. If the **fname** column contains the string `'Ludwigssss'`, the result is still `6`.
TEXT column	Returns number of bytes in a column, including trailing white-space characters.	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, minus any trailing white pace, 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 `'Ludwigssss'`, the result is still `6`.

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 **stores_demo** 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 example 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, one byte stores one character. For more information about the default code set, see "The Default Locale" on page 1-29.

cust_name	length	cust_num
Ludwig	6	101
Carole	6	102
Philip	6	103
Anthony	7	104
Raymond	7	105

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. LENGTH returns the number of bytes in its argument. This result might be different from the number of characters.

The next example assumes that the database that contains the **customer_multi** table has locale with a multibyte code set. Suppose that the user enters a SELECT statement with the LENGTH function to display **lname**, its length, and **customer_num** for the customer whose number is 199.

```
SELECT lname AS cust_name,
   length (fname) AS length, customer_num AS cust_num
   FROM customer_multi WHERE customer_num = 199
```

Suppose that **lname** for customer 199 consists of four characters:

```
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 example 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
aA1A2bB1B2	6	199

The OCTET_LENGTH Function

The OCTET_LENGTH function returns the number of bytes and generally includes trailing whitespace characters in the byte count. This SQL length function uses the definition of whitespace that the locale defines. OCTET_LENGTH returns the number of bytes in a character column, quoted string, host variable, or SPL variable. The actual behavior of OCTET_LENGTH 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 whitespace character. For simplicity, the Example column also assumes that the example strings consist of single-byte characters.

OCTET_LENGTH Argument	Behavior	Example
Quoted string	Returns number of bytes in string, including any trailing white-space characters.	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 whitespace characters. 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 whitespace. Value is the actual length, in bytes, of the character string, not the declared 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-space characters.	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 whitespace, 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 whitespace in the byte count, whereas LENGTH generally excludes trailing whitespace 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 four single-byte characters (the characters `abc` and one trailing space)	4
'A^{1}A^{2}B^{1}B^{2}'	A quoted string with two multibyte characters	4
'aA^{1}A^{2}bB^{1}B^{2}'	A quoted string with two single-byte and two 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 whitespace. For simplicity, the **Example** column assumes that the strings consist of single-byte characters.

CHAR_LENGTH Argument	Behavior	Example
Quoted string	Returns number of characters in string, including any trailing white-space (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 space characters. 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 whitespace characters. Value is the actual length, in bytes, of the string, not the declared 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 space characters.	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 space, regardless of declared 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 `'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 declared 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
'A1A2B1B2'	A quoted string with 2 multibyte characters	2
'aA1A2B1B2'	A quoted string with 2 single-byte and 2 multibyte characters	4

Using Locale-Sensitive Data Types

The previous section described how NCHAR and NVARCHAR data types are sorted in localized order, if the locale defines one. This section explains how a locale affects the way that a database server handles the MONEY data type, extended data types, and smart large objects (CLOB and BLOB data types).

For the syntax of these data types, see the *IBM Informix Guide to SQL: Syntax*. For descriptions of these data types, see the *IBM Informix Guide to SQL: Reference*.

Handling the MONEY Data Type

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.

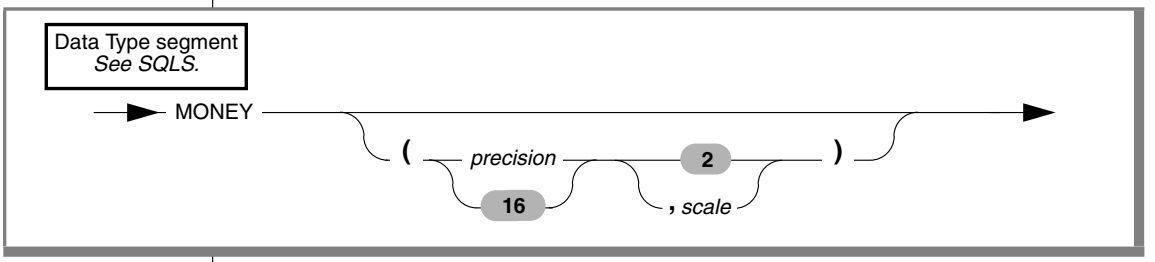
The choice of locale affects monetary data in the following ways:

- The default value of *scale* in the declaration 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-6.

Specifying Values for the Scale Parameter

Define a MONEY column with the following syntax.



Element	Purpose
precision	Total number of significant digits in a decimal or money data type You must specify an integer between `1` and `32`, inclusive. The default *precision* is `16`.
scale	Number of digits to the right of the decimal point.

The *scale* must be an integer between 1 and *precision*. If you omit the *scale*, the database server provides a default *scale* that the database locale defines. For the default locale (U.S. English), the default is 2, as the diagram indicates.

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. An example of a 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 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 data value in the column might be 123456.7890.

The GLS code sets for most European languages can support the euro symbol in monetary values. For the complete syntax of the MONEY data type, see the *IBM Informix Guide to SQL: Syntax*. For a complete description of the MONEY data type, see the *IBM 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 about currency notation, see "Numeric and Monetary Formats" on page 1-19.

In the default locale, the default currency symbol is a dollar sign ($), the default thousands separator is a comma (,), and the default decimal separator is a period (.) symbol. 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 1-48.

IDS

Handling Extended Data Types

The extensible data type system of Dynamic Server allows users to define new data types and the behavior of these new data types to the database server. This section explains how these types are handled in GLS processing. See also *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Opaque Data Types

An opaque data type is fully encapsulated to client applications; that is, 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. All GLS processing (code-set conversion, localized collation order, end-user formats, and so on) must be performed in the opaque-type support functions.

When you create an opaque data type, you can write the support functions as C UDRs that can handle any locale-sensitive data. For more information, see "Locale-Sensitive Data in an Opaque Data Type" on page 4-27.

Complex Data Types

Dynamic Server also supports complex data types:

- Collection data types: SET, MULTISET, and LIST
- Row data types: named ROW types and unnamed ROW types

Any of these data types can have members with character, time, or numeric data types. The database server can still handle the GLS processing for these data types when they are part of a complex data type.

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 opaque or built-in type, a named ROW type, or another distinct data type. Dynamic Server handles GLS considerations for a distinct type as it would for the source type.

IDS

Handling 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 recovery and transaction rollback. Dynamic Server supports two smart-large-object types:

- The BLOB data type stores any type of binary data, including images and video clips.
- The CLOB data type 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 search for information in smart large objects. Functions of the GLS library can assist you in this task. For more information, see the *IBM Informix GLS Programmer's Manual*.

To access smart large objects through a client application, you must use an API, such as ESQL/C or DataBlade API. Because GLS does not support direct access to smart-large-object data through SQL, GLS does not automatically handle the data (no automatic code-set conversion, localized collation order, end-user formats, and so on). All support must be done within an API.

When you copy CLOB data from a file, Dynamic Server performs any necessary character-set conversions. If the client or server locale (when it copies from client and server files, respectively) differs from the database locale, Dynamic Server invokes the routines to convert to the database locale.

Using Data Manipulation Statements

The choice of a locale can affect these SQL data manipulation statements:

- DELETE
- INSERT
- LOAD
- UNLOAD
- UPDATE

Sections describe the GLS aspects of these SQL statements. For a complete description of these statements, see the *IBM Informix Guide to SQL: Syntax*.

Specifying Conditions in the WHERE Clause

These statements can include a WHERE clause to specify rows to operate on:

- For the DELETE statement, the WHERE clause specifies rows to delete.
- For the INSERT statement with 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 3-33 and "Comparisons with MATCHES and LIKE Conditions" on page 3-37.

Specifying Era-Based Dates

These 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 1-36.

- If your locale does not support era-based 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 1-47.

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 the *IBM Informix Guide to SQL: Syntax*.

Loading Data into a Database

The LOAD statement inserts data from an operating-system file into an existing table or view. This operating-system 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, the 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 sending the 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 apply any format that the client locale supports to column values in the LOAD FROM file. For example, a French locale might define monetary values that have a blank space as the thousands separator and a comma as the decimal separator. When you use this locale, the following literal value for a MONEY column is valid in a LOAD FROM file:

```
3 411,99
```

You can specify alternative formats for date and monetary data. If you set appropriate 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 1-46 and "Customizing Monetary Values" on page 1-48.

Unloading Data from a Database

The UNLOAD statement writes the rows that a SELECT statement retrieves to an operating-system file. This operating-system file is called an UNLOAD TO file. The data values in this file contains characters that the client code set defines. If the client locale supports a multibyte code set, the data can include multibyte characters from the code set.

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 the 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 1-17 and "Customizing End-User Formats" on page 1-46.

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.*

XPS

Loading with External Tables

High-performance parallel loading and unloading for Extended Parallel Server uses *external tables*. It uses a series of enhanced SQL statements that you can issue with DB-Access or embed in ESQL/C.

The high-performance loader (HPL) provides extensive support for loading tables from many different sources, and performs a variety of data-format conversions. It also supports non-ASCII characters in field and record delimiters. High-performance loading performs operations like the following that might involve support for non-ASCII characters:

- Transfers data files across platforms with the Informix data format
- Transfers operational data from a mainframe to a data warehouse
- Uses the database server to convert data between delimited ASCII, fixed ASCII, EBCDIC, and Informix internal (raw) representation
- Uses INSERT and SELECT statements of SQL to specify the mapping of data to new columns in a database table

In nondefault locales, enhanced SQL statements for the loader, such as CREATE EXTERNAL TABLE...USING, INSERT INTO...SELECT, and SELECT...INTO EXTERNAL *table-name* USING, support identifiers that can include non-ASCII characters. For information about identifiers for Extended Parallel Server, see "Non-ASCII Characters in Identifiers" on page 3-5.

XPS

Loading Simple Large Objects with External Tables

Extended Parallel Server supports external tables to load and unload simple large objects. Simple large objects (TEXT or BYTE data type columns) are supported only by delimited and INFORMIX format external tables. In delimited format, a simple-large-object column can be represented in either text or hex encoding. In text encoding, a simple large object is written to data file as is. Backslashes and delimiters are escaped. In hex encoding, each data byte in a simple large object is represented by two hexadecimal digits (0 through 9, A through F, and all standard ASCII characters). Nonprintable characters in simple large objects are included unchanged in data files.

For information about how to define simple-large-object columns in an external table, see the CREATE EXTERNAL TABLE statement in the *IBM Informix Guide to SQL: Syntax*. For information on file formats and performance considerations, as well as a step-by-step procedure for loading with external tables, see the *Administrator's Reference*.

Specifying an Escape Character

You can specify an escape character to direct the database server to recognize incomplete or invalid multibyte character data in the simple large object. If you do not specify an escape character, the database server does not check the character fields in text-based data files for embedded special characters during loading.

When you specify an escape character, the backslash (\) character precedes any single character to indicate the occurrence of the actual character, regardless of whether it would otherwise have a special significance to the loading and unloading process. For example, '\|' is interpreted as the literal '|' character instead of as a column separator.

During unloading, the database server escapes delimiters and backslash (\) symbols. During loading, any character that immediately follows a backslash is taken literally. Nonprintable characters are directly embedded in the data file for TEXT column values.

Defining a Delimiter

Simple-large-object data values are inserted directly into the record at the point where the TEXT or BYTE column is defined, between field delimiters.

User-defined delimiters are limited to one byte each. Therefore, in multibyte locales, only characters with a length of exactly one byte can be defined as delimiters. In both single byte and multibyte locales, a simple large object is always traversed byte by byte. If a byte matches one of the delimiters or a backslash, it is escaped during unloading. During loading, only the byte immediately following a backslash is escaped, not the (possibly multibyte) character following the backslash.

Transversal of delimited simple-large-object data is performed byte by byte in all locales. A simple large object is not traversed character by (possibly multibyte) character because it does not always contain valid text, and might contain incomplete or invalid multibyte characters. Unlike character columns, blank filling or truncating for simple large objects is not an option for invalid multibyte characters. You cannot have random access to the data in simple large objects, and you cannot alter simple large objects in any way.

Important: *The database server does not detect incomplete or invalid multibyte characters in simple-large-object data in the loading or unloading process. You must ensure that multibyte data is consistent and accurate before you load it into a character column.*

Chapter

4

Database Server Features

In This Chapter

This chapter describes how the GLS feature affects the database server. It covers the following main topics:

- Which operating-system files the database server can access
- When the database server uses code-set conversion
- Which database server utilities provide support for the GLS feature

For more information about these database server features, see the *Administrator's Guide*. For more information about database server utilities, see the *Administrator's Reference*. For information about migrating to a different Informix database server, see the *IBM Informix Migration Guide*.

GLS Support by Informix Database Servers

The database server can perform read and write operations to the following operating-system files:

- Diagnostic files

  Diagnostic files include the following files:

  - **af.*xxx***
  - **shmem.*xxx***
  - **gcore.*xxx*** ♦ UNIX
  - core

  The database server generates diagnostic files when you set one or more of the following configuration parameters:

  - DUMPDIR
  - DUMPSHMEM
  - DUMPCNT
  - DUMPCORE
  - DUMPGCORE ♦ UNIX
- Message-log file

  The database server generates a user-specified message-log file when you set the MSGPATH configuration parameter.

These operating-system files reside on the server computer, where the database server resides. When the database server reads from or writes to these files, it must use a code set that the server computer supports. The database server obtains this code set from the server locale.

Set the server locale with the **SERVER_LOCALE** environment variable. If you do not set **SERVER_LOCALE**, the database server uses the default locale, as the server locale. For details, see "SERVER_LOCALE" on page 2-32.

For Extended Parallel Server, all coservers must have identical GLS operating-system environments. ♦ XPS

To perform code-set conversion and handle non-ASCII characters that are associated with read and write operations on operating-system files, the database server determines the database server code set (the code set that the database server locale supports). For information about the use of non-ASCII characters, see "Non-ASCII Characters in Identifiers" on page 3-5.

Database Server Code-Set Conversion

This section summarizes the code-set conversion that the database server performs. For more general information about code-set conversion, see "Performing Code-Set Conversion" on page 1-40.

An Informix database server automatically performs code-set conversion between the code sets of the server-processing locale and the server locale 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-processing locale and server locale.

For a list of files for which Informix database servers perform code-set conversion, see "GLS Support by Informix Database Servers" on page 4-4. For information on GLS code-set conversion files, see "Code-Set-Conversion Files" on page A-10.

Once the database 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 IBM 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 **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables are set so that the server-processing locale supports a code set with these non-ASCII characters. For more information about the server-processing locale, see "Determining the Server-Processing Locale" on page 1-36.

The database server checks the validity of a filename with respect to the server-processing locale before it references the filename.

XPS

Extended Parallel Server rejects any filename that includes any character that is not ASCII alphanumeric 7-bit. ♦

Data That the Database Server Converts

When the database 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, the database 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, the database 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 the database server used to create the file. For more information, see "Making Sure That Your Product Supports the Same Code Set" on page 2-15.
- If these two code sets are different, but no Informix code-set conversion exists, the database server cannot perform code-set conversion.

  If the database 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.

Locale-Specific Support for Utilities

This section provides information that is specific to the use of the GLS feature by database server utilities. For a complete description of utilities, see your *Administrator's Reference*.

IDS

For information about database server utilities for auditing, see the *IBM Informix Trusted Facility Guide.* ♦

Database server utilities and SQL utilities are client applications that request information from an instance of the 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 the database server 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 "Performing Code-Set Conversion" on page 1-40.

Windows

Changes to locale environment variables should also be reflected in the Windows registry database under HKEY_LOCAL_MACHINE. ♦

Non-ASCII Characters in Database Server Utilities

Most database 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).

The following table shows utilities that accept non-ASCII characters in command-line arguments or produce non-ASCII output.

Utility Name	Non-ASCII Characters in Command-Line Arguments	Non-ASCII Output
onaudit (IDS)	-f *input_file*	Yes
oncheck (IDS)	-cc -pc *database* -ci -cI -pk -pK -pl -pL *database*:*table*#*index_name* -ci -cI -pk -pK -pl -pL -cd -cD -pB -pt -pT -pd -pD -pp *database*:*table*	Yes
onload (IDS)	*database*:*table* -i *old_index new_index* -t *tape_device*	Yes
onlog (IDS)	-d *tape_device*	
onpload (IDS)	-d *source* -j *jobname* -p *projectname*	Yes
onshowaudit (IDS)	-f *input_file* -s *server_name*	Yes
onspaces (IDS)	-p *pathname* -f *filename*	

(1 of 2)

Utility Name	Non-ASCII Characters in Command-Line Arguments	Non-ASCII Output
onstat	-o *filename -dest* (IDS) *filename_source* (IDS) None (XPS)	Yes
onunload (IDS)	*database:table* -t *tape_device*	Yes
onutil (XPS)	CHECK TABLE DATA *database:owner:table* CHECK TABLE INFO *database:owner:table*	Yes

(2 of 2)

XPS

You can use **xctl**, the Extended Parallel Server control utility, to execute other database server utilities such as **onstat**. ♦

Non-ASCII Characters in SQL Utilities

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 the **chkenv** utility, refer to the *IBM Informix Guide to SQL: Reference*. For a description of the **dbload**, **dbschema, dbexport,** and **dbimport** utilities, refer to the *IBM Informix Migration Guide*. For information about DB-Access, see the *IBM Informix DB-Access User's Guide*.

The DB-Access utility generates labels and messages in the code set of the client locale.

XPS

For Extended Parallel Server, DB-Access accepts multibyte command-line arguments for *database* and *script_file*. ♦

IDS

DB API

Locale Support For C User-Defined Routines

Dynamic Server allows you to create user-defined routines (UDRs) that are written in the C programming language. These *C UDRs* use the DataBlade API to communicate with the database server. For a complete description of the DataBlade API, see the *IBM Informix DataBlade API Programmer's Guide*. This section describes how to *internationalize* a C UDR.

Internationalization is the process of creating a user-defined routine (UDR) that can support different languages, territories, and code sets without changing or recompiling its code. For a complete discussion of internationalization, see the *IBM Informix GLS Programmer's Manual*.

An internationalized C UDR must handle the following GLS considerations:

- Where can the UDR use non-ASCII characters in source code?
- What steps must the C UDR take when copying character data?
- How can the UDR access GLS locales?
- How does the UDR handle code-set conversion?
- How does the UDR handle locale-specific end-user formats?
- How can the UDR access internationalized exception messages?
- How can the UDR access internationalized tracing messages?
- How do opaque-type support functions handle locale-sensitive data?

Current Processing Locale for UDRs

To access a database, a client application first requests a connection to the database server, which must verify that it can access the specified database and establish the connection between the client and this database. In the process, the database server establishes the server-processing locale to use the duration of the connection. When the client application executes a UDR, this UDR executes on the server computer in the context of the server-processing locale. This locale is often called the *current processing locale*.

Many user-defined routines handle non-ASCII data correctly even if they were originally written for ASCII data. Some routines, however, might perform abnormally. To internationalize your C UDR, you must ensure that your UDR handles the server-processing locale in any GLS-related operations. If the UDR does not properly support the server-processing locale, the routine might return unexpected results or an error message.

Non-ASCII Characters in Source Code

Non-ASCII characters might appear in these contexts in a C-UDR source file:

- In C-language statements, such as variable declarations and **if** statements
- In SQL statements, which are sent to the database server through the **mi_exec( )** or **mi_exec_prepared_statement( )** functions

In C-Language Statements

The C compiler must recognize the code set that you use in your C-language statements. The capabilities of your C compiler might limit your ability to use non-ASCII characters within the C-language statements in a UDR source file. For example, some C-language compilers support multibyte characters in literals or comments only.

If the C compiler does not fully support non-ASCII characters, it might not successfully compile a UDR that contains these characters. In particular, the following situations might affect compilation of your UDR:

- 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 result in 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.

Tip: *The C compiler must also recognize the ASCII code set to be able to interpret the names of the DataBlade API functions within your C UDR.*

In SQL Statements

In C UDRs, SQL statements occur as literal strings to the **mi_exec( )** and **mi_prepare( )** functions. The C compiler does not parse these literal strings. Therefore, it does not need to recognize the code set of the characters in these SQL statements.

Within a C source file, you can use non-ASCII characters in SQL statements for the following objects:

- Names of SQL identifiers such as databases, tables, columns, views, constraints, prepared statements, and cursors

  For more information, see "Naming Database Objects" on page 3-3.

- Literal strings

  For example, in a UDR, the following use of multibyte characters is valid:

```
mi_exec(conn,
   "insert into tbl1 (nchr1) values 'A1A2B1B2'", 0);
```

- Filenames and pathnames, as long as your operating system supports multibyte characters in filenames and pathnames

Important: *To use non-ASCII characters in your SQL statements, your server-processing locale must include either a code set that supports these characters or a code set that is compatible with the character code set. For information on how to perform code-set conversion, see* *"Character Strings in UDRs" on page 4-15*.

Copying Character Data

When you copy data, you must ensure that the buffers are an 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 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 DataBlade API module performs the copy.

The IBM Informix GLS Library

The IBM Informix GLS library is an application programming interface (API) through which developers of user-defined routines and of DataBlade modules can create internationalized applications.

Character Processing with IBM Informix GLS

The macros and functions of IBM Informix GLS provide access within a DataBlade API module to GLS locales for culture-specific information. This library contains functions that provide the following capabilities:

- Process single-byte and multibyte characters
- Format date, time, and numeric data to locale-specific formats

For more information on the IBM Informix GLS library and how to use it in a DataBlade API module, see the *IBM Informix GLS Programmer's Manual*. If you do not have the printed manual, you can view online documentation at http://www-3.ibm.com/software/data/informix/pubs/library/.

Compatibility of Wide-Character Data Types

Wide character data types are an alternative form for the processing of multibyte characters. A wide-character form of a code set involves the normalization of the size of each multibyte character so that each character is the same size. A legacy DataBlade API module might use any of the following data types to hold wide characters.

Wide-Character Data Type	Description	Drawback
mi_wchar	A legacy DataBlade API data type currently defined as unsigned short on all systems	The DataBlade API does *not* provide wide-character functions that operate on **mi_wchar** values.
wchar_t	An operating-system data type that is platform-specific	The operating-system provides wide-character functions that operate on **wchar_t** values. Use of these functions is platform specific.

The IBM Informix GLS library provides the **gl_wchar_t** data type for support of wide characters. IBM Informix GLS also provides its own set of wide-character functions that operate on **gl_wchar_t**. Use of the IBM Informix GLS wide-character functions removes platform dependency from your application and provides access within your DataBlade API module to IBM Informix GLS locales.

The IBM Informix GLS library does *not* provide any functions for conversion between **gl_wchar_t** and **mi_wchar** or **gl_wchar_t** an **wchar_t**. If a DataBlade API module continues to use either **mi_wchar** or **wchar_t** and also needs to use the IBM Informix GLS wide-character processing, you must write code to perform any necessary conversions.

Code-Set Conversion and the DataBlade API

Within a UDR, the DataBlade API does *not* perform any code-set conversion automatically. Your C UDR might need to perform code-set conversion in the following situations:

- In strings that contain SQL statements
- In an opaque-type support function for an opaque type that contains character data

Character Strings in UDRs

When your C UDR contains character strings that are sent to the database server, it must perform any required code-set conversion on these strings. This code-set conversion must handle any differences between the code set of this character string and the code set of the server-processing locale in which the UDR executes.

For example, the DataBlade API does not perform code-set conversion on the multibyte table name, $A^1A^2A^3B^1B^2$, in the following SELECT statement:

```
mi_exec(conn, "SELECT * from A¹A²A³B¹B²", 0);
```

If your UDR might execute in a server-processing locale that does not include a code set that supports characters in your SQL statements, the UDR can explicitly perform code-set conversion between the code sets of the server-processing locale and a specified locale.

The DataBlade API provides the following functions to assist in this code-set conversion.

Code-Set Conversion on a String	DataBlade API Function
Perform code-set conversion on a specified string from a specified locale to the server-processing locale	**mi_convert_from_codeset( )**
Perform code-set conversion on a specified string from the server-processing locale to a specified locale	**mi_convert_to_codeset( )**

For more information on the syntax of these DataBlade API functions, see the function reference of the *IBM Informix DataBlade API Programmer's Guide*.

Character Strings in Opaque-Type Support Functions

The client application performs code-set conversion of non-opaque-type data that is transferred to and from the client, but the database server does not know about the internal format of an opaque data type. Therefore, for opaque data types, the support functions are responsible for explicitly converting any string that is not in the code set of the server-processing locale.

You might need to perform code-set conversion in the following opaque-type support functions:

- In the input and output support functions: to convert the external format of the opaque type between the code sets of the client locale and the server-processing-locale
- In the receive and send support functions: to convert any character fields in the internal structure of the opaque type

Tip: *The code that the DataBlade Developer's Kit (DBDK) generates for opaque-type input and output support functions handles external formats from nondefault locales.*

The DataBlade API provides the following functions for code-set conversion in the support functions of an opaque data type.

Code-Set Conversion on an Opaque Type	DataBlade API Function
Perform code-set conversion on a string argument from the code set of the server-processing locale to that of the client locale	**mi_put_string( )**
Perform code-set conversion on a string from the code set of the client locale to that of the server-processing locale	**mi_get_string( )**

For more information on the syntax of these DataBlade API functions, see the function reference in the *IBM Informix DataBlade API Programmer's Guide*.

Locale-Specific Data Formatting

When a C UDR handles strings that contain end-user formats for date, time, numeric, or monetary data, you must write the UDR so that it handles any locale-specific formats of these end-user formats. The DataBlade API provides functions that convert between the internal representation of several data types and its end-user format.

The following DataBlade API functions convert an internal database value to a string that uses the locale-specific end-user format.

DataBlade API Function	Description
mi_date_to_string( )	Uses the locale-specific end-user date format to convert an internal DATE value to its string equivalent.
mi_money_to_string( )	Uses the locale-specific end-user monetary format to convert an internal MONEY value to its string equivalent.
mi_decimal_to_string( )	Uses the locale-specific end-user numeric format to convert an internal DECIMAL value to its string equivalent.

Important: *The **mi_datetime_to_string( )** and **mi_interval_to_string( )** functions do not format strings in the date and time formats of the current processing locale. Instead, they create a date, time, or interval string in a fixed ANSI SQL format.*

The following DataBlade API functions interpret a string in its locale-specific end-user format and convert it to its internal database value.

DataBlade API Function	Description
mi_string_to_date( )	Converts a string in its locale-specific date end-user format to its internal DATE format.
mi_string_to_money( )	Converts a string in its locale-specific currency end-user format to its internal MONEY format.
mi_string_to_decimal( )	Converts a string in its locale-specific numeric end-user format to its internal DECIMAL format.

Important: *The **mi_string_to_datetime( )** and **mi_string_to_interval( )** functions do not interpret the date and time formats of the current processing locale. Instead, they interpret the date/time or interval string in a fixed ANSI SQL format.*

Internationalized Exception Messages

The DataBlade API function **mi_db_error_raise( )** sends an exception message to an exception callback. This message can be either of the following:

- A *literal message*, which you provide as the third argument to **mi_db_error_raise( )**
- A *customized message* that is associated with a value of **SQLSTATE**, which you provide as the third argument to **mi_db_error_raise( )**

  The **mi_db_error_raise( )** function can raise exceptions with customized messages, which DataBlade modules and UDRs can store in the **syserrors** system catalog table. The ***s*yserrors** table maps these messages to five-character **SQLSTATE** values. In the **syserrors** table, you can associate a locale with the text of a customized message.

For general information on how to specify a literal message in **mi_db_error_raise( )** and how to specify a customized message for **mi_db_error_raise( )**, see the chapter on how to handle exceptions and events in the *IBM Informix DataBlade API Programmer's Guide*.

This section discusses the following tasks about how to raise locale-specific exception messages:

- How to add a locale-specific exception message to the **syserrors** system catalog table
- How the choice of locale in a customized message affects the way that **mi_db_error_raise( )** searches for a customized message
- How to specify parameter markers that contain non-ASCII characters

Inserting Customized Exception Messages

You can store customized status codes and their associated messages in the **syserrors** system catalog table. To create a customized exception message, insert a row directly in the **syserrors** table. The **syserrors** table provides the following columns for an internationalized exception message.

Column Name	Description

sqlstate	The **SQLSTATE** value that is associated with the exception
	You can use the following query to determine the current list of **SQLSTATE** message strings in **syserrors**:
	`SELECT sqlstate, locale, message`
	`FROM syserrors`
	`ORDER BY sqlstate, locale`
	For more information on how to determine **SQLSTATE** values, see the *IBM Informix DataBlade API Programmer's Guide*.

message	The text of the exception message, with characters in the code set of the target locale By convention, do *not* include any newline characters in the message.
locale	The locale with which the exception message is to be used The **locale** column identifies the language and code set used for the internationalization of error and warning messages. This name is the name of the target locale of the message text.

For more information on the **syserrors** system catalog table, see the chapter that describes the system catalog in the "*IBM Informix Guide to SQL: Reference*." Do *not* allow any code-set conversion when you insert the text in **syserrors**.

If the code sets of the client and database locales differ, temporarily set both the **CLIENT_LOCALE** and **DB_LOCALE** environment variables in the client environment to the name of the database locale. This workaround prevents the client application from performing code-set conversion.

If you specify any parameters in the message text, include only ASCII characters in the parameter names, so s that the parameter name can be the same for *all* locales. Most code sets include the ASCII characters. For example, the following INSERT statements insert new messages in **syserrors** whose **SQLSTATE** value is "03I01":

```
INSERT INTO syserrors VALUES ("03I01", "en_us.8859-1", 0, 1,
      "Operation Interrupted.")
INSERT INTO syserrors VALUES ("03I01", "fr_ca.8859-1", 0, 1,
      "Traitement Interrompu.")
```

The '03I01' **SQLSTATE** value now has two locale-specific messages. The database server chooses the appropriate message based on the server-processing locale of the UDR when it executes. For more information on how **mi_db_error_raise( )** locates an exception message, see "Searching for Customized Messages" on page 4-22.

Inserting a Localized Exception Message from a C UDR

As noted in the previous section, when you create messages for exceptions raised within user-defined routines (UDRs) by **mi_db_error_raise( )**, the locale of the message text must match the server-processing locale. If these locales are different, then the use of an SQL script or of a C UDR that calls the **mi_exec( )** function to insert the message is not reliable, because the SQL parser issues an exception when it encounters characters that it does not recognize. To avoid this restriction, you can use a UDR that prepares the INSERT statement (with **mi_prepare( )**) to load the error messages:

- Use placeholders ('?' symbols) for the **SQLSTATE** value and the error-message text. These values are in the first and last columns (**sqlstate** and **message**, respectively) of the **syserrors** system catalog table.
- Hardcode the name of the locale that the message text uses. The locale name is in the second column (**locale**) of **syserrors**.

For example, the following line prepares an INSERT statement for messages in the default locale (**en_us**) on a UNIX system:

```
stmt = mi_prepare(conn,
   "insert into syserrors (?, 'en_us.8859-1', 0, 1, ?)", NULL);
```

When executing this statement, you must provide values for the placeholders (**sqlstate** and **message**) and then use the **mi_exec_prepared_statement( )** function to send the prepared INSERT statement to the database server.

The following UDR code uses a message array (**enus_msg**) to hold the SQLSTATE values and their associated message text. It puts information about each element of this message array in the appropriate placeholder arrays (**args**, **lens**, **nulls**, and **types**) of the **mi_exec_prepared_statement( )** function.

```
#include <stdio.h>
#include <string.h>
#include "mi.h"

#define MAX_MSG 3
char *enus_msg[MAX_MSG][2] =  {
   "XT010", "First error message for insertion",
   "XT020", "Second error message for insertion",
   "XT030", "Third error message for insertion"
    };

/*
 * Title: gls_insert_enus
 * Purpose: Add localized messages to 'syserrors' system error table
 *          for given locale, independent of session locale setting.
 */
mi_integer
gls_insert_enus()
{
MI_DATUM       args[2];                  /* pointers to column values */
mi_integer     lens[2];                  /* lengths of column values */
mi_integer     nulls[2];                 /* null capability of columns */
mi_string     *types[2];                 /* types of columns */
mi_integer     i;
MI_STATEMENT  *stmt;
MI_CONNECTION *conn = mi_open(NULL, NULL, NULL);

/*
 * Prepare statement using placeholder values for sqlstate and message
 * columns and fixed values for locale, level, and seqno columns.
 */
stmt = mi_prepare(conn,
         "insert into syserrors values(?,'en_us.8859-1',0,1,?)", NULL);
for (i=0; i<MAX_MSG; i++)                /* Loop through message array */
{
   args[0] = (MI_DATUM)enus_msg[i][0];   /* Set pointer to sqlstate string
*/
   lens[0] = strlen(args[0]);            /* Set length of sqlstate string
*/
   nulls[0] = MI_FALSE;                  /* Set null handling capability */
   types[0] = "char(5)";                 /* Set sqlstate column type */
   args[1] = (MI_DATUM)enus_msg[i][1];   /* Set pointer to message string
*/
   lens[1] = strlen(args[1]);            /* Set length of message string */
   nulls[1] = MI_FALSE;                  /* Set null handling capability */
   types[1] = "varchar(255)";            /* Set message column type */
   mi_exec_prepared_statement(stmt,0,0,2,args,lens,nulls,types,NULL,NULL);
}
mi_close(conn);
return 0;
}
```

For descriptions of executing prepared statements and of how to add customized messages to the **syserrors** system catalog table, see the *IBM Informix DataBlade API Programmer's Guide*.

Searching for Customized Messages

When the **mi_db_error_raise( )** function initiates a search of the **syserrors** system catalog table, it requests the message in which all components of the locale (language, territory, code set, and optional modifier) are the same in the current processing locale and the **locale** column of **syserrors**.

For C UDRs that use the default locale, the current processing locale is U.S. English (**en_us**). When the current processing locale is U.S. English, **mi_db_error_raise( )** looks *only* for messages that use the U.S. English locale. For C UDRs that use nondefault locales, however, the current processing locale is the server-processing locale.

For a description of how **mi_db_error_raise( )** searches for messages in the **syserrors** system catalog table, see the chapter on exceptions in the *IBM Informix DataBlade API Programmer's Guide*.

Specifying Parameter Markers

The customized message in the **syserrors** system catalog table can contain *parameter markers*. These parameter markers are strings of characters enclosed by a single percent (%) symbol on each end (for example, %TOKEN%). A parameter marker is treated as a variable for which the **mi_db_error_raise( )** function can supply a value. The **mi_db_error_raise( )** function assumes that any message text or message parameter strings that you supply are in the server-processing locale. For a complete description of how to specify parameter markers for a customized message, see the *IBM Informix DataBlade API Programmer's Guide*.

Internationalized Tracing Messages

The API supports trace messages that correspond to a particular locale. The current database locale determines which code set the trace message uses. Based on the current database locale, a given tracepoint can produce an internationalized trace message. Internationalized tracing enables you to develop and test the same code in many different locales.

To provide internationalized tracing support, the API provides the following capabilities:

- The **systracemsgs** system catalog table stores internationalized trace messages.
- Two internationalized trace functions, **gl_dprintf( )** and **gl_tprintf( )**, format internationalized trace messages.

Inserting Messages in the systracemsgs System Catalog Table

The **systracemsgs** system catalog table stores internationalized trace messages that you can use to debug your C UDRs. To create an internationalized trace message, insert a row directly into the **systracemsgs** table.

The **systracemsgs** table describes each internationalized trace message.

Column Name	Description
name	The name of the trace message
locale	The locale with which the trace message is to be used
message	The text of the trace message

The combination of message name and locale must be unique within the table. Once you insert a new trace class into **systracemsgs**, the database server assigns it a unique identifier, called a *trace-message identifier*. It stores the trace-class identifier in the **msgid** column of **systracemsgs**. Once a trace message exists in the **systracemsgs** table, you can specify the message either by name or by trace-message identifier to API tracing functions.

The trace-message text can be a string of text in the appropriate language and code set for the locale, and can contain *tokens* to indicate where to substitute a piece of text. Token names are delimited between percent (%) symbols. The following INSERT statement puts a new message called **qp1_exit** in the **systracemsgs** table:

```
INSERT INTO informix.systracemsgs(name, locale, message)
VALUES ('qp1_exit', 'en_us.8859-1',
   'Exiting msg number was %ident%; the input is still %i%')
```

This message text is in English and therefore the **systracemsgs** row specifies the default locale of U.S. English.

This second message is the French version of the **qp1_exit** message and therefore the **systracemsgs** row specifies a French locale on a UNIX system (**fr_fr.8859-1**):

```
INSERT INTO informix.systracemsgs(name, locale, message)
VALUES ('qp1_exit', 'fr_fr.8859-1',
   'Le numéro de message en sortie était %ident%; \
    l'entrée est toujours %i%')
```

Enter message text in the language of the server locale, with any characters available in the server code set. To insert a variable, enclose the variable name with a a single percent sign on each end (for example, **%a%**). When the database server prepares the trace message for output, it replaces each variable with its actual value.

Putting Internationalized Trace Messages into Code

The DataBlade API provides the following tracing functions to insert internationalized tracepoints into UDR code:

- The GL_DPRINTF macro formats an internationalized trace message and specifies the threshold for the tracepoint. The syntax for GL_DPRINTF is as follows:

```
GL_DPRINTF(trace_class, threshold,
   (message_name [,toktype, val]...,MI_LIST_END));
```

- The **gl_tprintf( )** function formats an internationalized trace message but does *not* specify a tracepoint threshold.

  The **gl_tprintf( )** function is for use within a trace block, which uses the **tf( )** function to compare a specified threshold with the current trace level. The syntax for **gl_tprintf( )** is as follows:

```
gl_tprintf(message_name [,toktype ,val]...,
   MI_LIST_END);
```

Syntax elements for both GL_DPRINTF and **gl_tprintf( )** have these values:

trace_class	is either a trace-class name or the trace-class identifier integer value expressed as a character string.
threshold	is a nonnegative integer that sets the tracepoint threshold for execution.
message_name	is the identifier for an internationalized message stored in the **systracemsgs** system catalog table of the database.
toktype	is a string made up of a token name followed by a single percent (%) symbol followed by a single character output specifier as used in **printf** formats.
val	is a value expression to be output that must match the type of the output specifier in the preceding token.
MI_LIST_END	is a macro constant that ends the variable-length list.

Important: *The MI_LIST_END constant marks the end of the variable-length list. If you do not include MI_LIST_END, the user-defined routine might fail.*

This internationalized trace statement uses the GL_DPRINTF macro:

```
i = 6;
/* If the current trace level of the funcEntry class is greater
 * than or equal to 20, find the version of the qp1_entry
 * message whose locale matches the current database locale
 */
GL_DPRINTF("funcEntry", 20,
         ("qp1_entry",
         "ident%s", "one",
         "i%d", i,
         MI_LIST_END));
```

In the default locale, if the current trace level of the **funcEntry** class is greater than or equal to 20, this tracepoint generates the following trace message:

```
13:21:51  Exiting msg number was one; the input is still 6
```

The following internationalized trace block that uses the **gl_tprinf()** function:

```
i = 6;
/* Compare current trace level of "funcEnd" class and
 * with a tracepoint threshold of 25. Continue execution of
 * trace block if trace level >= 25
 */
if ( tf("funcEnd", 25) )
   {
      i = doSomething();
   /* Generate an internationalized trace message (based
    * on current database locale) */
      gl_tprintf("qp1_exit", "ident%s", "deux", "i%d", i,
      MI_LIST_END);
   }
```

If the locale is French and the current trace level of the **funcEntry** class is greater than or equal to 25, the tracepoint generates this trace message:

```
13:21:53 Le numéro de message en sortie était deux; l'entrée est toujours 6
```

The database server writes the trace messages in the trace-output file in the code set of the locale associated with the message. If the trace message originated from the **systracemsgs** system catalog table, its characters are in the code set of the locale specified in the **locale** column of its **systracemsgs** entry. The database server might have performed code-set conversion on these trace messages if the code set in the UDR source is different from (but compatible with) the code set of the server-processing locale.

Searching for Trace Messages

To write an internationalized trace message to your trace-output file, the database server must locate a row in the **systracemsgs** system catalog table whose **locale** column matches (or is compatible with) the server-processing locale for your UDR. Therefore, to see a particular trace message in the trace-output file, environment variables that specify the locale (**CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE**) must be set so that the database server generates a server-processing locale that matches an entry in the **systracemsgs** system catalog table.

The database server searches the **systracemsgs** table for an entry with the same name as the tracepoint and a locale in which all components of the locale (language, territory, and code set) are the same in the current processing locale and the **locale** column of **systracemsgs**. If only the language and territory match, the database server converts the code set. If no message has matching language and territory, it uses the first available message with the correct language. If there is no message in the appropriate language, it uses the message for the default language, **en_us**.

Locale-Sensitive Data in an Opaque Data Type

An opaque data type is fully encapsulated. Its internal structure is not known to the database server. The database server cannot automatically perform the locale-specific tasks such as code-set conversion on character data or locale-specific formatting of date, numeric, or monetary data. 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 DataBlade API and IBM Informix GLS provide GLS support for opaque-type support functions written in C. The following sections summarize GLS considerations for these support functions. For general information on the support functions of an opaque data type, see *IIBM Informix User-Defined Routines and Data Types Developer's Guide*.

Internationalized Input and Output Support Functions

The internal representation of an opaque data type is the C structure that stores the opaque-type data. Each opaque type also has a character-based format, known as its *external representation*, which is received by the database server as an LVARCHAR value. This can hold single-byte (ASCII and non-ASCII) and multibyte character strings, depending on the locale of the client application. (The data length of an LVARCHAR external representation is limited only by the operating system, not by the 32,739 byte maximum size of LVARCHAR columns in Dynamic Server databases.)

Client applications perform code-set conversion on LVARCHAR data types. The ability to transfer the data between a client application and database server, however, is not sufficient to support locale-sensitive data in opaque data types. It does not ensure that data values are correctly manipulated at the destination.

The **input( )** and **output( )** support functions convert the opaque data type from its internal to an external representation, and vice versa, as follows:

- 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.

Opaque-type support functions written as C UDRs must ensure that these functions correctly handle any locale-sensitive data, including these tasks.

Locale-Sensitive Task	For More Information
Any code-set conversion on character data	"Code-Set Conversion and the DataBlade API" on page 4-15
Any handling of multibyte or wide characters in character data	"The IBM Informix GLS Library" on page 4-13
Any formatting of locale-specific date, numeric, or monetary data	"Locale-Specific Data Formatting" on page 4-17

Internationalized 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:

- The **receive( )** function converts the internal representation of the opaque data type on the client computer to its internal representation on the server computer.
- The **send( )** function converts the internal representation of the opaque data type on the client computer to its internal representation on the server computer.

If the internal representation contains character data, the client application cannot perform any locale-specific translations, including these.

Locale-Sensitive Task	For More Information
Any code-set conversion on character data	"Character Strings in Opaque-Type Support Functions" on page 4-16
Any handling of multibyte or wide characters in character data	"The IBM Informix GLS Library" on page 4-13

When you write **receive( )** and **send( )** support functions as C UDRs, you must ensure that these functions handle these locale-sensitive tasks correctly.

Chapter

5

General SQL API Features

E/C

In This Chapter

This chapter explains how the GLS feature affects applications that you develop with the IBM Informix Client Software Developer's Kit. This chapter includes the following sections:

- "Supporting GLS in IBM Informix Client Applications"
- "Internationalizing Client Applications"
- "Handling Locale-Specific Data"

Supporting GLS in IBM Informix Client Applications

To connect to a database, an ESQL/C 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/C program performs this step automatically when it requests a connection.
- Checks for connection warnings that the database server generates

  You must include code in your ESQL/C program to perform this step.

Client Application Code-Set Conversion

This section summarizes the code-set conversion that a client product performs. For more general information about code-set conversion, see "Performing Code-Set Conversion" on page 1-40.

The client application automatically performs code-set conversion between the client and database code sets when both of these conditions are true:

- The code sets of the client and database locales 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 the **CLIENT_LOCALE** setting).

If the client and database code sets are the same, no code-set conversion is needed. If the code sets do not match, however, 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.

On UNIX, you can use the **glfiles** utility to obtain a list of code-set conversions that your IBM Informix product supports. For more information, see "The glfiles Utility" on page A-16. On Windows, you can examine the directory **%INFORMIXDIR%\gls\cv*Y*** to determine the GLS code-set conversions that your IBM Informix product supports. For more information on this directory, see "Code-Set-Conversion Files" on page A-10.

If no code-set-conversion files exist, the client application generates a run-time error when it starts up to indicate incompatible code sets. If 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 assumes that:

- All data values that is processes are 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 the eighth character field of the SQLWARN array for a 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 Windows Code Page 1252 (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 the eighth character field of the SQLWARN array to `W` 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

Your application, however, might 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 is still valid because both database locales support the default ISO8859-1 code set.

Instead, if the application opens a database with the Japanese SJIS (**ja_jp.sjis**) locale, the database server sets the SQLWARN warning 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. When the client application started, it determined that code-set conversion was required between the Windows Code Page 1252 and ISO8859-1 code set. The client application performs this code-set conversion until it terminates.

When you open a database with **ja_jp.sjis**, the client application would perform code-set conversion incorrectly because the code sets are different. It would continue to convert between Windows Code Page 1252 and ISO8859-1 instead of between Windows Code Page 1252 and Japanese SJIS. This situation could lead to corruption of data.

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 6-24*.

Data That a Client Application Converts

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

IDS

- Values of SQL data types
  - CHAR, VARCHAR, NCHAR and NVARCHAR
  - TEXT (the BYTE data type is *not* converted)
  - LVARCHAR
  - Character data in opaque data types (if their support functions perform the code-set conversions) ♦
- Values of ESQL/C character types (**char**, **fixchar**, **string**, and **varchar**)
- SQL statements, both static and dynamic
- SQL identifiers. These include names of columns, tables, views, prepared statements, cursors, constraints, indexes, triggers, and other database objects. For a list of SQL objects that can include non-ASCII characters in their identifiers, see "Non-ASCII Characters in Identifiers" on page 3-5.
- SPL text
- Command text
- Error message text in the **sqlca.sqlerrm** field

Internationalizing Client Applications

To internationalize or localize a client application, use IBM Informix GLS, an application programming interface (API) for applications that use a C-language interface. For more information, see "GLS Support by IBM Informix Products" on page 1-6 and the *IBM Informix GLS Programmer's Manual*.

Internationalization

Internationalization is the process of creating or modifying an application so that it can use the correct GLS locale to support different languages, territories, and code sets without changing or recompiling the code.

This process makes IBM Informix database applications easily adaptable to any culture and language. For a database application, you perform internationalization on the application that accesses a database, not on the database. The data in a database that the application accesses should already be in a language that the end user can understand.

To internationalize a database application, design the application so that the tasks in the following table do not make any assumptions about the language, territory, and code set that the application uses at runtime.

Application Task	Description
User interfaces	Includes any text that is visible to end users, including menus, buttons, prompts, help text, status messages, error messages, and graphics
Character processing	Includes the following processing tasks: ■ Character classification ■ Character case conversion ■ Collation and sorting ■ Character versus byte processing ■ String traversal ■ Code-set conversion

(1 of 2)

Application Task	Description
Data formatting	Includes any culture-specific formats for numeric, monetary, date, and time values
Documentation	Includes any explanatory material such as printed manuals, online documentation, and README files
Debugging via tracing (IDS, DB API)	The DataBlade API provides the application or DataBlade developer the capability of using internationalized trace messages. It uses in-line code working with system catalog tables: **systracemsgs** and **systraceclasses**. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

(2 of 2)

An internationalized application dynamically obtains language-specific information for these application tasks. Therefore, one executable file for the application can support multiple languages.

Localization

Localization is the process of adapting a product to a specific cultural environment. This process usually involves the following tasks:

- Creating culture-specific resource files
- Translating message or resource files
- Setting date, time, and money formats
- Translating the product user interface

Localization might also include the translation and production of end-user documentation, packaging, and collateral materials.

To localize a database application, you create a database application for a specific language, territory, and code set. Localization involves the following tasks:

- Ensure that GLS locales exist for the desired language, territory, and code set.
- Translate the character strings in any external resource or message files that the application uses.

Important: *An internationalized application is much easier localize than a non-internationalized application.*

Choosing a GLS Locale

To localize your application, choose a locale that provides the culture-specific information for the language, territory, and code set that the application is to support. For information about locales, see "Setting a GLS Locale" on page 1-21.

An internationalized application makes no assumptions about how these locales are set at runtime. Once the application environment specifies the locales to use, the application can access the appropriate GLS locale files for locale-specific information. As long as a GLS locale is provided that supports a particular language, territory, and code set, the application can obtain the locale-specific information dynamically.

The *current processing locale* (sometimes called just the *current locale*) is the locale that is currently in effect for an application. It is based on one of the following environments:

- The client environment

  ESQL/C creates client applications. Therefore, the current processing locale for ESQL/C applications is the client locale.
- The database that the database server is currently accessing

DB API

The current processing locale for DataBlade client applications is the client locale. The current processing locale for DataBlade UDRs is the server-processing locale, which the database server determines from the client, database, and server locales. ♦

Translating Messages

An internationalized application should not have any language-specific text within the application code. This language-specific text includes the following kinds of strings:

- Strings that the application displays or writes

  Examples include error messages, informational messages, menu items, and button labels.

- Strings that the application uses internally

  Examples include constants, filenames, and literal characters or strings.
- Strings that an end user is expected to enter

  Examples include `yes` and `no` responses.

Tip: *You do not need to put SQL keywords (such as SELECT, WHERE, INSERT, and CREATE) in a message file. In addition, language keywords (such as* ***if****,* ***switch****,* ***for****, and* ***char****) do not need to appear in a message file.*

In an internationalized application, these strings appear as references to external files, called *resource files* or *message files*. To localize these strings of the database application, you must perform the following tasks:

- Translate all strings within the external files.

  The new external files contain the translated versions of the strings that the application uses.
- Set the **DBLANG** environment variable to the subdirectory within **INFORMIXDIR** that contains the translated message files that the IBM Informix products use.

  The **INFORMIXDIR** environment variable indicates the location where the IBM Informix products are installed. You can use the **rgetmsg( )** and **rgetlmsg( )** functions to obtain IBM Informix product messages. For more information on these functions, see the *IBM Informix ESQL/C Programmer's Manual*.

Handling Locale-Specific Data

Each IBM Informix SQL API product contains a processor to process an ESQL/C source file that has embedded SQL and preprocessor statements. The ESQL/C processor, **esql**, processes C source files.

The processors for ESQL/C products use operating-system files in the following situations:

- They write language-specific source files (**.c**) when they process an ESQL/C source file.

  The ESQL/C processors use the client code set (that the client locale specifies) to generate the filenames for these language-specific files.
- They read ESQL/C source files (**.ec**) that the user creates.

  The ESQL/C processors use the client code set to interpret the contents of these ESQL/C source files.

Use the **CLIENT_LOCALE** environment variable to specify the client locale.

Processing Characters

A GLS locale supports a specific code set, which can contain single-byte characters and multibyte characters. When your application processes only single-byte characters, if can perform string-processing tasks based on the assumption that the number of bytes in a buffer equals the number of characters that the buffer can hold. For single-byte code sets, you can rely on the built-in scaling for array allocation and access that the C compiler provides.

If your application processes multibyte characters, however, it can no longer assume that the number of bytes in a buffer equals the number of characters in the buffer. Because of the potential of varying number of bytes for each character, you can no longer rely on the C compiler to perform character-processing tasks such as traversing a multibyte-character string and allocating sufficient space in memory for a multibyte-character string.

You can use functions from the IBM Informix GLS library to communicate to your application how to perform internationalization on character-processing tasks.

Character-processing tasks include the following:

- String traversal
- String processing
- Character classification
- Case conversion
- Character comparison and sorting

For more information and the syntax of these functions, see the *IBM Informix GLS Programmer's Manual*.

Formatting Data

When you internationalize an application, consider how to handle the format of locale-specific data. The format in which numeric, monetary, and date and time data appears to the end user is locale specific. The GLS locale file defines locale-specific formats for each of these types of data, as the following table shows.

Type of Data	Locale-File Category
Numeric	NUMERIC
Monetary	MONETARY
Date and Time	TIME

The IBM Informix GLS library provides functions that allow you to perform the following tasks on locale-specific data:

- *Conversion* changes a string that contains locale-specific format to the internal representation of its value

  You usually perform conversion on a locale-specific string to prepare it for storage in a program variable or a database column.
- *Formatting* changes the internal representation of a value to locale-specific string.

  You usually perform formatting of a locale-specific string to prepare the internal representation of a value for display to the end user.

For more information and the syntax of these functions, see the *IBM Informix GLS 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 are as follows:

- 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 3-24.

Copying Character Data

When you copy data, you must ensure that the buffers are an 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 **A1A2A3B1B2B3** 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 **A1A2A3B1B2**. 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. If the client and database locales are different and convertible, the application might need to expand this value. For more information, see "Performing Code-Set Conversion" on page 1-40.

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 a 10-byte buffer to be adequate to hold an 8-byte character value from the database. If the client application expands this value to 12 bytes, however, the value no longer fits in the **buf1** buffer. The **fname** value is truncated to fit in **buf1**, possibly creating partial characters if **fname** contains multibyte characters. For more information, see "Partial Characters in Column Substrings" on page 3-24.

To avoid this situation, define buffers to handle the maximum character-expansion possible, 4 bytes, in the conversion between your client and database code sets.

Chapter

6

IBM Informix ESQL/C Features

E/C

In This Chapter

This chapter explains how the GLS feature affects ESQL/C, an SQL application programming interface (API). It includes the following sections:

This chapter covers GLS information that is specific to ESQL/C. For additional GLS information for ESQL/C, see Chapter 5, "General SQL API Features."

Tip: *For features that are not unique to the GLS feature, see the "IBM Informix ESQL/C Programmer's Manual." For a discussion of the IBM Informix GLS library, a set of C language functions, procedures, and macros that allow you to develop internationalized applications, see the "IBM Informix GLS Programmer's Manual." For information about the DataBlade API, a C language API that is provided with Dynamic Server, see the "IBM Informix DataBlade API Programmer's Guide."*

Handling Non-ASCII Characters

The ESQL/C processors obtain the code set for use in ESQL/C source files from the client locale. Within an ESQL/C source file, you can use non-ASCII characters for the following program objects:

- ESQL/C comments
- Names of SQL identifiers such as databases, tables, columns, views, constraints, prepared statements, and cursors

  For more information, see "Naming Database Objects" on page 3-3.

- ESQL/C host variable and indicator variable names

  For example, in an ESQL/C program, this 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 "Using Non-ASCII Characters in Host Variables" on page 6-4.

- 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';
  ```

IDS

- Filenames and pathnames, if your operating system supports multibyte characters in filenames and pathnames. ♦

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 "Generating Non-ASCII Filenames" on page 6-6.*

To use non-ASCII characters in your ESQL/C source file, the client locale must support them. For information about the use of non-ASCII characters, see "Non-ASCII Characters in Identifiers" on page 3-5.

Using 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 "Setting a GLS Locale" on page 1-21.
- 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 6-8.

ESQL/C applications can also support non-ASCII characters in comments and SQL identifiers. For more information, see "Non-ASCII Characters in Identifiers" on page 3-5.

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';
```

If 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
```

Generating Non-ASCII Filenames

When an ESQL/C source file is processed, the ESQL/C processor generates a corresponding intermediate file for the source file. If you use non-ASCII characters (8-bit and multibyte character) in these source filenames, the following restrictions affect the ability of the ESQL/C processor to generate filenames that contain non-ASCII characters:

- The ESQL/C processor must know whether the operating system is 8-bit clean.

  For more information, see "GLS8BITFSYS" on page 2-13.
- The code set of the client locale (the client code set) must support the non-ASCII characters that are used in the ESQL/C source filename.
- Your C compiler supports the non-ASCII characters that the filename of the ESQL/C 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 "Generating Non-ASCII Filenames" on page 6-6.

Using Non-ASCII Characters in ESQL/C Source Files

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.

The capabilities of your C compiler, however, 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 that 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 some single-byte characters such as percent (%), comma (,), backslash (\ ), and double quote (") characters. If such characters are included 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 $A^1A^2A^3$ 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:

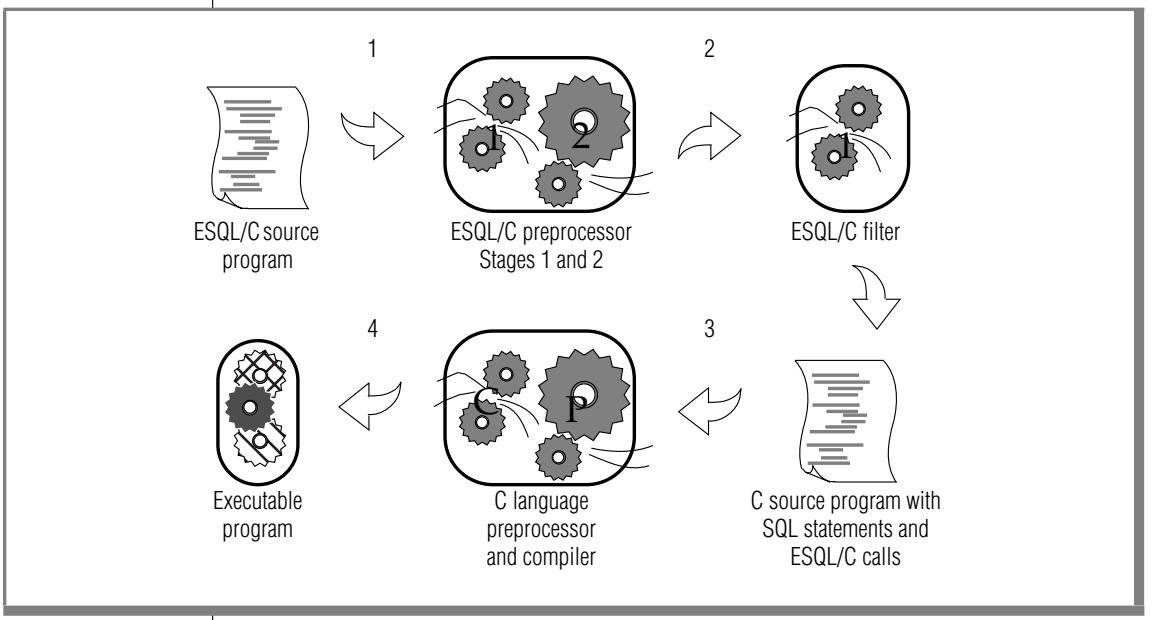
```
stcopy("A1"A3, dest); /* incorrect interpretation of A2 */
```

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 6-1 shows how an ESQL/C program that contains non-ASCII characters becomes an executable program.

Figure 6-1
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.

- The **CC8BITLEVEL** environment variable indicates the 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	esqlmf Action
0	Converts all non-ASCII characters, in literal strings *and* comments, to octal constants.
1	Converts non-ASCII characters in literal strings, but not in 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**, then **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.

Defining Variables for Locale-Sensitive Data

The SQL data types NCHAR and NVARCHAR support locale-specific data, in the sense that the database server uses localized collation (if the locale defines localized collation), rather than code set order, for sorting data strings of these types. For more information about NCHAR and NVARCHAR 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. You can use these four host-variable data types for NCHAR and NVARCHAR data.

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, note 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 3-12 and "The NVARCHAR Data Type" on page 3-14.

You can insert a value that a character host variable (**char**, **fixchar**, **string**, or **varchar**) holds in columns of the NCHAR or NVARCHAR data types.

Using Enhanced ESQL/C Library Functions

IBM Informix SQL API products support locale-specific enhancements to the ESQL/C library functions. These ESQL/C 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/C functions might produce.

DATE-Format Functions

The ESQL/C DATE-format functions are as follows:

- **rdatestr( )**
- **rstrdate( )**
- **rdefmtdate( )**
- **rfmtdate( )**

These functions support extensions to format era-based DATE values:

- Support for the **GL_DATE** environment variable
- Era-based date formats of the **DBDATE** environment variable
- Extensions to the date-format strings for ESQL/C DATE-format functions
- Support for a precedence of date end-user formats

This section describes locale-specific behavior of the ESQL/C DATE-format functions. For details, see the *IBM Informix ESQL/C Programmer's Manual*.

GL_DATE Support

The **GL_DATE** setting les can affect the results that these ESQL/C 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 6-15.

DBDATE Extensions

The ESQL/C DATE-format functions that support the extended era-based date syntax for the **DBDATE** environment variable are as follows:

- **rdatestr( )**
- **rstrdate( )**

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 prints this value for the date 08/18/1990:

```
H02/08/18
```

Important: *IBM Informix products treat any undefined characters in the alphabetic era specification as an error.*

If you set **DBDATE** to a era-based date format (which 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

The ESQL/C DATE-format functions that support the extended-DATE format strings are as follows:

- **rdefmtdate( )**
- **rfmtdate( )**

The following table shows the extended-format strings that these ESQL/C functions support for use with GLS locales. These extended-format strings format eras with 2-digit year offsets.

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 (which **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 (which **CLIENT_LOCALE** indicates) defines

The following table shows some extended-format strings for era-based dates. These examples assume that the client locale is Japanese SJIS (**ja_jp.sjis**).

Description	Example Format	October 5, 1990 prints as:
Abbreviated era year	gyymmdd	H021005
	gyy.mm.dd	H02.10.05
Full era year	eyymmdd	A1A2021005
	eyy-mm-dd	A1A202-10-05
	eyyB^{1}B^{2}mmB^{1}B^{2}ddB^{1}B^2	A1A202B1B210B1B205B1B2

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);
stcopy("gyy/mm/dd", fmt_str);
rdefmtdate(&jdate, fmt_str, in_str);
printf("Abbreviated Era Year: %s\n", in_str);

stcopy("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
```

Precedence for Date End-User Formats

The ESQL/C 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 end-user date format that the client locale specifies (if **CLIENT_LOCALE** is set)
4. The end-user date format from the default locale: **%m %d %iY**

For more information on the precedence of **DBDATE**, **GL_DATE,** and **CLIENT_LOCALE**, refer to "Date and Time Precedence" on page 1-47.

Tip: *IBM Informix products support **DBDATE** for compatibility with earlier products. It is recommended that you use the **GL_DATE** environment variable for new client applications.*

DATETIME-Format Functions

The ESQL/C DATETIME-format functions are as follows:

- **dtcvfmtasc( )**
- **dttofmtasc( )**

These functions support extensions to format era-based DATETIME values:

- Support for the **GL_DATETIME** environment variable
- Support for era-based date and times of the **DBTIME** environment variable
- Extensions to the date and time format strings for ESQL/C DATETIME-format functions
- Support for a precedence of DATETIME end-user formats

This section describes locale-specific behavior of the ESQL/C DATETIME-format functions. For general information about the ESQL/C DATETIME-format functions, see the *IBM Informix ESQL/C Programmer's Manual*.

GL_DATETIME Support

The **GL_DATETIME** setting can affect results that these ESQL/C DATETIME-format functions generate. The end-user format that **GL_DATETIME** specifies overrides date and time formats that the client locale defines. For more information, see "Precedence for DATETIME End-User Formats" on page 6-17.

DBTIME Support

The ESQL/C DATETIME-format functions support the extended era-based date and time format strings for the **DBTIME** environment variable. When you set **DBTIME** to an era-based format, these functions can convert between literal DATETIME strings and internal DATETIME values.

Tip: *IBM Informix products support **DBTIME** for compatibility with earlier products. It is recommended that you use the **GL_DATETIME** environment variable for new applications.*

If you set **DBTIME** to a era-based DATETIME format (which 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/C DATETIME-format functions support.

Format	Description	December 27, 1991 Printed
%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`

(1 of 2)

Format	Description	December 27, 1991 Printed
%Y %m %dj1	Japanese era with abbreviated era symbols	`H0003 12 27`
%y %m %dj2	Japanese era with full era symbols	`A1A2B1B203 12 27`
%Y %m %dj2	Japanese era with full era symbols	`A1A2B1B20003 12 27`

(2 of 2)

In addition to the formats in the preceding table, these ESQL/C DATETIME-format functions support the GLS date and time specifiers. For a list of these specifiers, see "GL_DATE" on page 2-16 and "GL_DATETIME" on page 2-25.

Precedence for DATETIME End-User Formats

The ESQL/C 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 "Date and Time Precedence" on page 1-47.

Numeric-Format Functions

The ESQL/C numeric-format functions are as follows:

- **rfmtdec( )**
- **rfmtdouble( )**
- **rfmtlong( )**

These functions 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

This section describes locale-specific behavior of the ESQL/C numeric-format functions. For general information about the ESQL/C numeric-format functions, see the *IBM Informix ESQL/C Programmer's Manual*.

Tip: *For a list of errors that these ESQL/C numeric-format functions might return, see "GLS-Specific Error Messages" on page 6-23.*

Support for Multibyte Characters

The ESQL/C numeric-format functions support multibyte characters in their format strings if your client locale supports a multibyte code set that defines these characters. These ESQL/C functions and routines, however, 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 A^1A^2 in the format string:

```
stcopy("A1A2***,***", fmtbuf);
rfmtlong(78941, fmtbuf, outbuf);
printf("Formatted value: %s\n", outbuf);
```

This code fragment generates the following output (if the client code set contains the A^1A^2 character):

```
Formatting value: A1A2*78,941
```

Locale-Specific Numeric Formatting

The ESQL/C numeric-format functions 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 information about the locale-specific behavior of **DBMONEY**, see "DBMONEY Extensions" on page 6-22.

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 6-20. For more information on the MONETARY and NUMERIC locale categories, see "Locale Categories" on page A-3.

3. The actual symbol that appears in the format string ($, comma, or period)

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 logical 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/C numeric-format functions support all formatting characters that the *IBM Informix ESQL/C Programmer's Manual* describes. 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 *front* currency symbol if the locale defines one. The MONETARY category of the locale defines the *front* currency symbol, which is the symbol that appears before a monetary value. When you group several dollar signs in a row, a single currency symbol floats to the right-most position that it can occupy without interfering with the number.
@	This character is replaced by the *back* currency symbol if the locale defines one. The MONETARY category of the locale defines the *back* currency symbol, the symbol that appears after a monetary value.

For more information, see "The MONETARY Category" on page A-6.

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 *front* currency symbol and sets the *back* 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 *back* currency symbol and sets the *front* currency symbol to a blank character.

The default locale defines the currency symbol as the *front* currency symbol, which appears as a dollar sign ($). In the default locale, the *back* currency symbol appears as a blank space. 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, FF is the French currency symbol for French francs, and DM is the German currency symbol for deutsche marks.

The **DBMONEY** environment variable can also set the precede-currency symbol and the succeed-currency symbol. The syntax diagram in "DBMONEY" on page 2-10 refers to these symbols as *front* and *back*, respectively. The **DBMONEY** setting, if one is specified, takes precedence over the symbols that the MONETARY category of 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 if your client code set supports them. For example, the following table shows how multibyte characters appear in examples of output.

Format String	Number to Format	DBMONEY	Output
"$$,$$$.$$"	1234	'$'.	`$1,234.00`
"$$,$$$.$$"	1234	DM,	`DM1.234,00`
"$$,$$$.$$"	1234	A^1A^2.	`A1A21,234.00`
"$$,$$$.$$"	1234	.A^1A^2	`s1,234.00`
"&&,&&&.&&@"	1234	.A^1A^2	`s1,234.00A1A2`
"$&&,&&&.&&@"	1234	A^1A^2.	`A1A2s1,234.00`
"$&&,&&&.&&@"	1234	.A^1A^2	`s1,234.00A1A2`
"@&&,&&&.&&"	1234	.A^1A^2	`A1A2s1,234.00`

In the preceding table, the character s represents a blank or space.

String Functions

The following ESQL/C string functions support locale-specific shifted characters:

- **rdownshift( )**
- **rupshift ( )**

These string functions use the information in the CTYPE category of the client locale to determine the shifted code points. 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 before. You must ensure that the buffer you pass to these ESQL/C shift functions is large enough to accommodate this expansion.*

GLS-Specific Error Messages

The following ESQL/C functions might generate GLS-specific error messages:

- DATE-format functions
- DATETIME-format functions
- Numeric-format functions

For more information on GLS-specific error messages, use the **finderr** utility on UNIX or the **Informix Error Messages** utility on Windows.

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, check that the application correctly handles the following situations:

- When the application writes simple large objects (TEXT or BYTE data) to the database, it must set the **loc_type** field in the locator structure **loc_t** to indicate the type of simple large object that it needs to write.
- IDS
- When the application writes smart large objects (CLOB or BLOB data) to the database, it uses various large-object file descriptors. ♦
- When the application uses the **sqlda** structure to describe dynamic SQL statements, it must account for possible size differences in character data.
- 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 5-13.

Writing TEXT Values

ESQL/C uses the **loc_t** locator structure to read simple large objects from and write simple large objects to the database server. The **loc_type** field of this structure indicates the data type of the simple large object 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 simple large object data type implicitly. The database server assumes character data in the TEXT data type and noncharacter data in the BYTE data type.

If the client and database code sets are different and convertible, however, the client application must know the data type of the simple large object in order to determine whether to perform code-set conversion on the data.

Before an ESQL/C client application inserts a simple large object in the database, it must explicitly set the **loc_type** field of the simple large object:

- For a TEXT value, 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 data before it sends this data to the database for insertion.
- For a BYTE value, 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 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. To use these constants, you must include this header file in your ESQL/C source file.*

Your ESQL/C source code does not need to set **loc_type** before it reads simple-large-object data from a database. The database server obtains the data type of the simple large object 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 simple large object. 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 data 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 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 data. 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 **sqlda** structure is a dynamic-management structure that contains information about columns in dynamic SQL statements. The DESCRIBE...INTO statement uses the **sqlda** structure to return information about the columns in the select list of the Projection clause of a SELECT statement. It sets the **sqlvar** field of an **sqlda** structure to point to a sequence of partially filled **sqlvar_struct** structures. Each structure describes a single select-list column.

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 and 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 values when they are 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 the following code fragment shows:

```
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);
```

A buffer-size factor (BUFSIZE_FACT) of 4 is suggested 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 columns in the select list of the Projection clause at runtime. DESCRIBE returns the data type of a select-list column in the appropriate field of the dynamic-management structure that you use.

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 *IBM Informix Guide to SQL: Syntax*.

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 of data type CHAR or VARCHAR, then DESCRIBE returns the data type of the trimmed column as SQLVCHAR.
- If the source character-value expression is data type NCHAR or NVARCHAR, then DESCRIBE returns the data type of the trimmed column as SQLNVCHAR.

IDS

TRIM does not support the LVARCHAR data type. ♦

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 and use the TRIM function, 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. To use these constants, include this header file in your ESQL/C source file.*

Appendix

A

Managing GLS Files

This appendix describes the files provided for GLS, which are executable only. The following sections describe how to manage GLS files:

UNIX

Accessing GLS Files

IBM Informix products access the following GLS files to obtain locale-related information. For an overview of what type of information these files provide, see "Understanding a GLS Locale" on page 1-10.

GLS Files	Reference
GLS locale files	page A-3
Code-set-conversion files	page A-10
Code-set files	page A-13
The **registry** file	page A-14

In general, you do not need to examine the GLS files. You might, however, wish 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**): COLLATION category	page A-5
Exact code-set collation order	Source code-set file (***.cm**)	page A-13
Character mappings:		
Locale-specific mapping between uppercase and lowercase characters	Source locale file (***.lc**): CTYPE category	page A-4
Locale-specific classification of characters	Source locale file (***.lc**): CTYPE category	page A-4
Code-set-specific character mappings	Source code-set file (***.cm**)	page A-13
Mappings between characters of the source and target code sets	Source code-set-conversion file (***.cv**)	page A-10
Method for character mismatches during code-set conversion	Source code-set-conversion file (***.cv**)	page A-10
Code points for characters	Source code-set file (***.cm**)	page A-13
End-user formats:		
Numeric (nonmonetary) data	Source locale file (***.lc**): NUMERIC category	page A-6
Monetary data	Source locale file (***.lc**): MONETARY category	page A-6
Date data	Source locale file (***.lc**): TIME category	page A-7
Time data	Source locale file (***.lc**): TIME category	page A-7

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 describes the locale categories and the locations of the locale files.

Locale Categories

A locale file specifies behaviors for the locale categories. 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. For more information about end-user formats, see "End-User Formats" on page 1-17 and "Customizing End-User Formats" on page 1-46. The following table describes the locale categories and the behaviors for the default locale, U.S. English.

Locale Category	Description	In Default Locale (U.S. English)
CTYPE	Controls the behavior of character classification and case conversion.	The default code set classifies characters. On UNIX, the default code set is ISO8859-1. On Windows, the default code set is Windows Code Page 1252.
COLLATION	Controls the behavior of string comparisons.	The default locale does not define a localized order. Therefore, the database server collates NCHAR and NVARCHAR data in code-set order (unless SET COLLATION has specified some localized order)..
NUMERIC	Controls the behavior of non-monetary numeric end-user formats.	The following numeric notation for use in numeric end-user formats: ■ 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

(1 of 2)

Locale Category	Description	In Default Locale (U.S. English)
MONETARY	Controls the behavior of currency end-user formats.	The following currency notation for use in monetary end-user formats: ■ Currency symbol: dollar sign ($) appears as the *front* symbol before the currency value ■ No *back* currency symbol is defined. ■ Thousands separator: comma (,) ■ Decimal separator: period (.) ■ Number of digits between thousands separators: 3 ■ Symbol for positive number: plus (+) ■ Symbol for negative number: minus (-) Default scale for MONEY columns: 2
TIME	Controls the behavior of date and time end-user formats.	The following date and time end-user formats: ■ DATE values: **%m/%d/%iy** ■ DATETIME values: **%iY-%m-%d %H:%M:%S** No definitions for era-based dates.
MESSAGES	Controls the definitions of affirmative and negative responses to messages.	None

(2 of 2)

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 symbols.

This category might also include mappings between uppercase and lowercase letters. IBM Informix products access this category when they need to determine the validity of an identifier name, to shift the letter case of a character, or to compare characters.

The COLLATION Category

The COLLATION category can define a localized order. When an IBM 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 supports the following capabilities:

- Multicharacter collation elements define sets of characters that the database server should collate as a single unit. For example, the localized collating order might treat the Spanish double-L (ll) as a single collation element instead of as a pair of l's.
- Equivalence classes assign the same collation weight to different 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, IBM Informix products use code-set order for collation of all character data types:

- CHAR
- LVARCHAR ♦ (IDS)
- NCHAR
- NVARCHAR
- TEXT
- VARCHAR

The SET COLLATION statement can specify a localized collation that is different from the COLLATION setting of the locale that **DB_LOCALE** specifies. The scope of the collating order that SET COLLATION specifies is the current session, but database objects that can sort strings, such as constraints, indexes, UDRs, and triggers, always use the collating order from the time of their creation when they sort NCHAR or NVARCHAR values. ♦ (IDS)

The NUMERIC Category

The NUMERIC category defines the following numeric notation for end-user formats of nonmonetary numeric values:

- The numeric decimal separator
- The numeric thousands separator
- The number of digits to group together before inserting a 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.

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 information about alternative digits, see "Alternative Date Formats" on page 2-20 and "Alternative Time Formats" on page 2-27.

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 these characters (before or after)
- The scale (the number of fractional digits to the right of the decimal point) to display

This currency notation applies to the end-user formats of data from MONEY columns within a client application.

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 values of 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 "Specifying Values for the Scale Parameter" on page 3-49.

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 specifies the calendar (Gregorian, Hebrew, Arabic, Japanese Imperial, and so on) for reading or printing a month, day, or year.

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 2-20 and "Alternative Time Formats" on page 2-27.

This date and time information applies to the end-user formats of data in DATE and DATETIME columns within a client application.

Important: *Information in the TIME category does not affect the internal format of the DATE and DATETIME data types in the database.*

The MESSAGES Category

The MESSAGES category defines the format for affirmative and negative responses. This category is optional. IBM 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 IBM Informix product needs to obtain locale-specific information, it accesses one of the GLS locale files in the following table.

Platform	Locale File
UNIX	**$INFORMIXDIR/gls/lc*X*/*lg_tr*/*codemodf*.lco**
Windows	**%INFORMIXDIR%\gls\lc*X**lg_tr**codemodf*.lco**

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM Informix product, and **gls** is the subdirectory that contains the GLS files. This rest of this section describes the remaining elements in the pathname of GLS locale files.

Locale-File Subdirectories

The subdirectories of the **lc*X*** 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.

The next table shows some languages and territories that IBM Informix products can support, and their associated locale-file subdirectory names.

Language	Territory	Locale-File Subdirectory
English	Australia	**en_au**
	United States	**en_us**
	Great Britain	**en_gb**
German	Germany	**de_de**
	Austria	**de_at**
	Switzerland	**de_ch**
French	Belgium	**fr_be**
	Canada	**fr_ca**
	Switzerland	**fr_ch**
	France	**fr_fr**

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.

  IBM 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 locale. A section of the locale source file supports each of the locale categories, unless that category is empty, as the next table shows.

Locale Category	Reference	Locale Category	Reference
CTYPE	page A-4	MONETARY	page A-6
COLLATION	page A-5	TIME	page A-7
NUMERIC	page A-6	MESSAGES	page A-7

Locale Filenames

To conform to the *8.3 filename.ext* restriction on the maximum number of characters in valid filenames and file extensions on DOS systems, 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 next table shows some code sets and locale modifiers that IBM Informix products can support, along with their associated locale source filenames.

Code Set	Locale Modifier	Locale Source File
ISO8859-1 (IBM CCSID 819)	*None*	**0333.lc**
	Dictionary	0333dict.lc
Windows Code Page 1252 (West Europe)	*None*	**04e4.lc**
	Dictionary	04e4dict.lc
IBM CCSID 850	*None*	**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 Windows Code Page 1252, both the **fr_fr** and **en_us** locale-file subdirectories contain a **04e4.lc** locale file as well.

Other GLS Files

In addition to GLS locale files, IBM 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 **registry** file converts locale aliases to valid locale filenames. ♦

Windows

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. IBM 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 1-33*.

When an IBM Informix product needs to obtain code-set-conversion information, it accesses one of the GLS code-set-conversion files in the following table.

Platform	Code-Set-Conversion File
UNIX	**$INFORMIXDIR/gls/cv*Y*/*code1code2*.cvo**
Windows	**%INFORMIXDIR%\gls\cv*Y**code1code2*.cvo**

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM 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 rest of this section describes the remaining elements in the pathname of GLS code-set-conversion files.

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.

  IBM 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

IBM Informix products use the 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.

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 code set (or ISO8859-1 with a hexademical 0333 code number) requires the following two code-set-conversion files:

- The **01b50333.cv** file describes the mappings to use when IBM 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 IBM 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 IBM 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 "Performing Code-Set Conversion" on page 1-40.

The following table shows some of the code-set conversions that IBM 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	Windows Code Page 1252	**033304e4.cvo**
Windows Code Page 1252	ISO8859-1	**04e40333.cvo**
ISO8859-1	IBM CCSID 850	**03330352.cvo**
IBM CCSID 850	ISO8859-1	**03520333.cvo**
Windows Code Page 1252	IBM CCSID 850	**04e40352.cvo**
IBM CCSID 850	Windows Code Page 1252	**035204e4.cvo**

Code-Set Files

A *code-set file* (also called a character-mapping or *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 that it supports. In addition, IBM Informix products can perform code-set conversion between the code sets that have code-set files.

When an IBM Informix product needs to obtain code-set information, it accesses one of the GLS code-set files in the following table.

Platform	Code-Set File
UNIX	**$INFORMIXDIR/gls/cm*Z*/*code*.cmo**
Windows	**%INFORMIXDIR%\gls\cm*Z**code*.cmo**

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM 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.

Windows

The Informix registry File

The 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. IBM Informix products use code-set numbers to determine the filenames of locale and code-set-conversion files.

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, IBM Informix products use the 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 IBM 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 IBM Informix product needs to obtain information about locale aliases, it accesses the GLS code-set files in the following path:

```
%INFORMIXDIR%\gls\cmZ\registry
```

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM 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 Code-Set Name-Mapping file,* ***registry,*** *from the Informix directory. Do not modify this file. IBM Informix products use this file for the language processing of all locales.*

Removing Unused Files

An IBM 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**)

Removing Locale and Code-Set-Conversion Files

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. 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 **lc***X* subdirectory in your Informix installation.

  For more information on the **lc*X*** pathname, see "Locale-File Subdirectories" on page A-8.

- *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 **cv*Y*** subdirectory in your Informix installation.

  For more information on the **cv*Y*** pathname, see "Code-Set-Conversion Filenames" on page A-12.

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 Code-Set Name-Mapping file,* ***registry****. IBM Informix products use these files for the language processing of all locales.*

Because IBM Informix products do not access source versions of locale and code-set conversion files, you can safely remove them. These files, however, provide useful online documentation for the supported locales and code-set conversions. If you have enough disk space, it is recommended 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

The source version of code-set files (**.cm**) are provided as online documentation for the locales and code-set conversions that use them. Because IBM Informix products do not access source code-set files, you can safely remove them. However, if you have enough disk space, it is recommended 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 MS-DOS 8.3 standard format for filenames, IBM 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 list of the following GLS-related files:

- The GLS locales that are available on your system
- The IBM Informix code-set-conversion files that are available on your system
- The IBM Informix code-set files that are available on your system

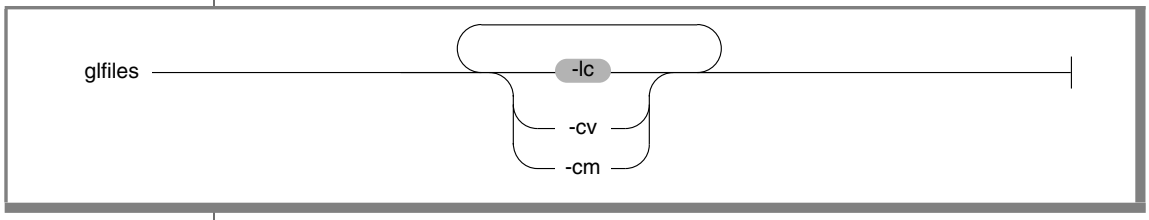
Before you run **glfiles**, take the following steps:

- Set the **INFORMIXDIR** environment variable to the directory in which you install your IBM Informix product.

  If you do not set **INFORMIXDIR**, **glfiles** checks the **/usr/informix** directory for the GLS files.
- Change 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.



Element	Purpose
-cv	The **glfiles** utility creates a file that lists the available code-set-conversion files.
-lc	The **glfiles** utility creates a file that lists the available GLS locales.
-cm	The **glfiles** utility creates a file that lists the available character mapping (charmap) files.

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 **cv*Y*** subdirectory in **$INFORMIXDIR/gls**, **glfiles** creates a file in your current directory that is called **cv*Y*.txt**, where ***Y*** is the version number of the code-set-conversion object-file format. The **cv*Y*.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/cv*Y*** 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-10.

Figure A-1 shows a file, **cv9.txt**, that lists 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-1
Sample glfiles File for Informix Code-Set-Conversion Files

Examine the **cv*Y*.txt** file to determine the code-set conversions that the **$INFORMIXDIR/gls/cv*Y*** directory on your system supports.

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 **lc*X*** subdirectory in the **gls** directory specified in **INFORMIXDIR**, **glfiles** creates a file in the current directory that is called **lc*X*.txt**, where ***X*** is the version number of the locale object-file format. The **lc*X*.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

Examine the **lc*X*.txt** files to determine the GLS locales that the **$INFORMIXDIR/gls/lc*X*** directory on your system supports.

Windows

To find out which GLS locales are available on your Windows system, you must look in the GLS system directories. A GLS locale resides in this file:

```
%INFORMIXDIR%\gls\lcX\lg_tr\codemodf.lco
```

In this path, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM Informix product, **gls** is the subdirectory that contains the GLS system files, ***X*** represents the version number of the locale file format, **lg** is the two-character language name, **tr** is the two-character territory name that the locale supports, and ***codemodf*** is the condensed locale name. ♦

Listing Character-Mapping Files

When you specify the **-cm** command-line option, the **glfiles** utility creates a file that lists the available character mapping (**charmap**) files. For each **cm*Z*** subdirectory in **$INFORMIXDIR/gls**, **glfiles** creates a file in the current directory that is called **cm*Z*.txt**, where ***Z*** is the version number of the **charmap** object-file format. The **cm*Z*.txt** file lists the character mappings in alphabetical order, sorted on the name of the GLS object **charmap** file.

Figure A-3 shows a sample file, **cm3.txt**, that contains the available character mappings.

```
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-3
Sample glfiles File for Informix Character-Mapping Files

Examine the **cm*Z*.txt** file to determine the character mappings that the **$INFORMIXDIR/ gls/cm*Z*** directory on your system supports.

Appendix

B

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

Index

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Symbols