Informix Product Family
Informix Global Language Support
Version 5.00

# *IBM Informix GLS API Programmer's Guide*

SC27-3849-01

Informix Product Family
Informix Global Language Support
Version 5.00

# *IBM Informix GLS API Programmer's Guide*

SC27-3849-01

**Note**
Before using this information and the product it supports, read the information in "Notices" on page C-1.

This edition replaces SC27-3849-00.

# Contents

# Introduction

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

## About this publication

This publication describes the global language support (GLS) application programming interface (API) available in IBM® Informix® ESQL/C and IBM Informix DataBlade® modules.

Application programmers use the IBM Informix GLS API to write programs (or change existing programs) to handle different languages, cultural conventions, and code sets.

### Types of users

This publication is written for DataBlade module developers and ESQL/C programmers who want to internationalize their applications with IBM Informix GLS.

This publication assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming

You can access the Informix information centers and other technical information such as technotes, white papers, and IBM Redbooks® publications online at http://www.ibm.com/software/data/sw-library/.

### Software compatibility

For information about software compatibility, see the IBM Informix GLS release notes.

### Assumptions about your locale

IBM Informix products can support many languages, cultures, and code sets. All the information related to character set, collation and representation of numeric data, currency, date, and time that is used by a language within a given territory and encoding is brought together in a single environment, called a Global Language Support (GLS) locale.

The IBM Informix OLE DB Provider follows the ISO string formats for date, time, and money, as defined by the Microsoft OLE DB standards. You can override that default by setting an Informix environment variable or registry entry, such as **DBDATE**.

If you use Simple Network Management Protocol (SNMP) in your Informix environment, note that the protocols (SNMPv1 and SNMPv2) recognize only English code sets. For more information, see the topic about GLS and SNMP in the *IBM Informix SNMP Subagent Guide*.

The examples in this publication are written with the assumption that you are using one of these locales: en_us.8859-1 (ISO 8859-1) on UNIX platforms or en_us.1252 (Microsoft 1252) in Windows environments. These locales support U.S. English format conventions for displaying and entering date, time, number, and currency values. They also support the ISO 8859-1 code set (on UNIX and Linux) or the Microsoft 1252 code set (on Windows), which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

You can specify another locale if you plan to use characters from other locales in your data or your SQL identifiers, or if you want to conform to other collation rules for character data.

For instructions about how to specify locales, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

### Demonstration databases

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

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix publications are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** 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 are in the `$INFORMIXDIR/bin` directory on UNIX platforms and in the `%INFORMIXDIR%\bin` directory in Windows environments.

## What's new in GLS API, Version 5.00

This publication includes information about new features and changes in existing functionality.

For a complete list of what's new in this release, see the release notes or the information center at http://publib.boulder.ibm.com/infocenter/idshelp/v117/topic/com.ibm.po.doc/new_features.htm.

*Table 1. What's new in IBM Informix GLS API Programmer's Guide for Version 5.00.xC7*

| Overview | Reference |
|---|---|
| Scan strings with the **ifx_gl_complen()** function<br><br>The **ifx_gl_complen()** function scans input strings faster than using the **ifx_gl_mblen()** function alone. The **ifx_gl_complen()** function returns the length in bytes of the initial part of an input string that matches a collating element, or returns `0` if the initial part of the string is not a collation sequence. | "The ifx_gl_complen() function" on page 4-5 |

*Table 2. What's new in IBM Informix GLS API Programmer's Guide for Version 5.00.xC1*

| Overview | Reference |
|---|---|
| New editions and product names<br><br>IBM Informix Dynamic Server editions were withdrawn and new Informix editions are available. Some products were also renamed. The publications in the Informix library pertain to the following products:<br>• IBM Informix database server, formerly known as IBM Informix Dynamic Server (IDS)<br>• IBM OpenAdmin Tool (OAT) for Informix, formerly known as OpenAdmin Tool for Informix Dynamic Server (IDS)<br>• IBM Informix SQL Warehousing Tool, formerly known as Informix Warehouse Feature | For more information about the Informix product family, go to http://www.ibm.com/software/data/informix/. |

# Character-representation conventions

Throughout this publication, examples show how single-byte and multibyte characters are displayed. Multibyte characters are usually ideographic (such as Japanese or Chinese characters), but this publication 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 publication represents multibyte and single-byte characters abstractly

## Single-byte characters

This publication 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 `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 six-character string abstractly:

```
abcdef
```

**Tip:** The letter “s” does not show in examples that represent strings of single-byte characters. The publication reserves the letter “s” as a symbol to represent a single-byte white space character. See also “White space characters in strings.”

## Multibyte characters

This publication 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 nth byte of the multibyte character, where n has values 2 - 4. For example, the following symbols represent a multibyte character that consists of 2 bytes:

$A^1A^2$

The following notation represents a multibyte character that consists of 4 bytes (the maximum length of a multibyte character):

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

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

CREATE DATABASE $A^1A^2B^1B^2C^1C^2D^1D^2E^1E^2$;

This statement creates a database whose name consists of five multibyte characters, each of which is 2 bytes long. For more about using multibyte characters in SQL identifiers, see Name database objects.

## 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 publication 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 2 bytes each:

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

## White space characters in strings

*White space* is a series of one or more characters that show as blank space, Each GLS locale defines what characters are white space characters.

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

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

s

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

ss

The following notation represents a multibyte white space character:

$s^1...s^n$

Here $s^1$ represents the first byte of the white space character, and $s^n$ represents the last byte of the white space character, where n can range 2 - 4. The following notation represents one 4-byte white space character:

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

### Trailing white space characters

Combinations of characters with white space 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 contains 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 white space characters) in a multibyte code set where each of the data characters and white space characters consists of 2 bytes:

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

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

$abcss^1s^2sss^1s^2$

The string has three single-byte characters (abc), a single-byte white space character (s), a multibyte white space character ($s^1s^2$), two single-byte white space characters (ss), and one multibyte white space character ($s^1s^2$).

## Example code conventions

Examples of SQL code occur throughout this publication. Except as 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 the code in 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 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. If you are using DB–Access, you must delimit multiple statements with semicolons.

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

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

## Additional documentation

Documentation about this release of IBM Informix products is available in various formats.

You can access or install the product documentation from the Quick Start CD that is shipped with Informix products. To get the most current information, see the Informix information centers at ibm.com®. You can access the information centers and other Informix technical information such as technotes, white papers, and IBM Redbooks publications online at http://www.ibm.com/software/data/sw-library/.

## Compliance with industry standards

IBM Informix products are compliant with various standards.

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 IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

The IBM Informix Geodetic DataBlade Module supports a subset of the data types from the *Spatial Data Transfer Standard (SDTS)—Federal Information Processing Standard 173*, as referenced by the document *Content Standard for Geospatial Metadata*, Federal Geographic Data Committee, June 8, 1994 (FGDC Metadata Standard).

## How to provide documentation feedback

You are encouraged to send your comments about IBM Informix user documentation.

Use one of the following methods:

- Send email to docinf@us.ibm.com.
- In the Informix information center, which is available online at http://www.ibm.com/software/data/sw-library/, open the topic that you want to comment on. Click the feedback link at the bottom of the page, fill out the form, and submit your feedback.
- Add comments to topics directly in the information center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

Feedback from all methods is monitored by the team that maintains the user documentation. The feedback methods are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Technical Support at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

# Chapter 1. Using Informix GLS

The IBM Informix GLS library enables a program to access the culture-specific information in the GLS locales that Informix provides.

This section provides information about the following topics:

- How the Informix GLS library can help you create internationalized programs.
- How to use the Informix GLS library
- How to optimize programs that use the Informix GLS library

For an introduction to GLS concepts and use, see the *IBM Informix GLS User's Guide*.

## Internationalized programs with Informix GLS

You can make IBM Informix database applications easily adaptable to any culture and language.

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 run time. The data in a database that the application accesses should already be in a language that the end user can understand.

*Table 1-1. Tasks for a database application*

| 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.<br>• Includes translating the character strings in any external resource or message files that the application uses. |
| Character processing | Includes the following processing tasks:<br>• Character classification<br>• Character case conversion<br>• Collation and sorting<br>• Character versus byte processing<br>• String traversal<br>• Code-set conversion |
| Data formatting | Includes any culture-specific formats for the following types of data:<br>• Numeric<br>• Monetary<br>• Date<br>• Time |
| Documentation | Includes any explanatory material such as printed manuals, online documentation, and readme files. |

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

## What is Informix GLS?

IBM Informix GLS is an application programming interface (API) for DataBlade module developers and ESQL/C programmers.

You can use the library of C language functions and macros of Informix GLS to write programs (or change existing programs) to handle different languages, cultural conventions, and code sets.

Of the tasks in Table 1-1 on page 1-1, the Informix GLS library facilitates the following:

**Character processing**
The Informix GLS library provides functions that perform the following character-processing tasks:

- Process single-byte, multibyte, and wide characters and strings
- Collate single-byte and multibyte characters and strings
- Process input and output
- Convert between compatible code sets

**Data formatting**
The Informix GLS library also provides functions that convert date, time, monetary, and numeric strings from and to internal representations.

Of the tasks in Table 1-1 on page 1-1, the Informix GLS library does not provide functions for the following:

**User interfaces**
You must ensure that user interfaces are designed in a way that is not language specific. In addition, put language-specific text, menus, buttons, and messages in external files that the application can reference at run time.

**Documentation**
You must ensure that any documentation you provide is translated to the correct language for a particular locale and territory. You might want to develop rules for the creation of documentation that simplify the translation tasks.

**Important:** Although the Informix GLS library does provide functions to perform case conversions (from lowercase to uppercase and vice versa), it does not provide a way to translate words, nor does it translate data in a database. You must obtain accurate and appropriate translations of text that the end user sees. These tasks must be performed by people experienced in the application and the languages involved.

## Informix GLS compatibility

The IBM Informix GLS library is compatible with most Informix products.

In order for you to use Informix GLS, both the database server and the application must support the following Informix application products:

- Informix ESQL/C develops client applications in the C language and can contain embedded SQL statements.

For more information about how to write ESQL/C applications, see the *IBM Informix ESQL/C Programmer's Manual*.

- The Informix DataBlade API supports development of client applications and user-defined routines (UDRs).

  Both kinds of DataBlade programs use DataBlade API functions to communicate with IBM Informix. For more information about how to use the DataBlade API, see the *IBM Informix DataBlade API Programmer's Guide*.

An ESQL/C or DataBlade API client application that uses Informix GLS can obtain locale information from an Informix database server that supports GLS functionality.

A DataBlade server routine can use Informix GLS to obtain locale information from IBM Informix with UD Option.

## Choose a GLS Locale

An IBM Informix GLS *locale* is a set of Informix files that bring together information about data that is specific to a particular culture, language, or territory.

In particular, a GLS locale provides the following information:

- The name of the code set that the application data uses
- The classes of characters in the code set
- The collation order to use for character data
- The format for date, time, numeric, and monetary data to appear to end users

Choose a locale that provides the culture-specific information for the language, territory, and code set that the application is to support. Locale files are located in various subdirectories under the `gls` directory of the `INFORMIXDIR` directory.

**Important:** Informix defines and preprocesses these locale files. You cannot modify Informix locales.

At run time, a database application uses the following locales.

**Client locale**

The locale that the database application uses for locale-specific information.

The application uses the client locale to obtain:

- The code set for read and write (I/O) operations on the client computer.
- The format for literal data strings (data formats).
- The code set for embedded SQL statements, host variables, and data sent to or received from the database server.

**Database locale**

The locale that the database uses for its data.

The application uses the database locale to obtain the code set for:

- Data sent to or received from the database server.
- Names of database objects, such as databases, tables, columns, and views.

If this code set is different from the client code set, the application must perform code-set conversion.

**Server locale**

The locale that the database server uses for locale-specific information.

When the client application establishes a connection, the database server uses the client locale, database locale, and server locale to determine the server-processing locale for this connection. The database server uses the server-processing locale for its own internal sessions and for access to any database that the connection opens.

At run time, an application obtains the locales that it needs from its application environment. You establish the locales of the application environment as follows.

| Locale | Default locale | Nondefault locale |
|---|---|---|
| Client locale | U.S. English | Set the **CLIENT_LOCALE** environment variable to the name of the locale you want. |
| Database locale | U.S. English | Set the **DB_LOCALE** environment variable to the name of the locale you want. |
| Server locale | U.S. English | Set the **SERVER_LOCALE** environment variable to the name of the locale you want. |

At run time, an internationalized program makes no assumptions about how these locales are set. Once the program environment specifies the locales to use, the application can access the correct GLS locale files for locale-specific information. As long as Informix provides a GLS locale that supports a particular language, territory, and code set, the program can obtain the locale-specific information dynamically.

The *current processing locale* (sometimes called the *current locale*) is the locale 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 current processing locale for DataBlade client applications is the client locale.
- The database that the database server is currently accessing

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

For more information about the default locale, client locale, database locale, server locale, server-processing locale, or any of the locale environment variables, see the *IBM Informix GLS User's Guide*.

## Using Informix GLS in a C-language program

To use IBM Informix GLS in a C-language program, include the following header file in the source file:

```
#include <ifxgls.h>
```

This section provides additional information about how to use the Informix GLS library.

**Related reference**:

"Compile and link Informix GLS"

Chapter 4, "Informix GLS functions," on page 4-1

# Compile and link Informix GLS

The following table lists the directories that must be accessible to compile and link an ESQL/C or DataBlade program with IBM Informix GLS.

| Contents | Windows directory | UNIX directory |
|---|---|---|
| Subdirectories for GLS locale and code-set conversion files | `%INFORMIXDIR%\gls` | `$INFORMIXDIR/gls` |
| Static and shared GLS libraries | `%INFORMIXDIR%\lib\esql` | `$INFORMIXDIR/lib/esql` |
| Two GLS header files: `gls.h` and `ifxgls.h` | `%INFORMIXDIR%\incl\public` | `$INFORMIXDIR/incl/public` |

In all these directories, the **INFORMIXDIR** environment variable is set to the directory where your Informix products are installed.

**Related reference**:

"Using Informix GLS in a C-language program" on page 1-4

## Informix GLS in ESQL/C applications

To compile and link ESQL/C applications that use IBM Informix GLS, issue the following command: **% esql source_file**

The **esql** preprocessor automatically links the Informix GLS library to an ESQL/C application. It also links the version of the GLS libraries (shared or static) that you specify with command-line options. For more information about how to compile an ESQL/C application, see the *IBM Informix ESQL/C Programmer's Manual*.

## Informix GLS in DataBlade programs

The way to compile and link a DataBlade program depends on whether the program is a DataBlade client application or a DataBlade user-defined routine (UDR).

**DataBlade client applications:**

You can compile and link a DataBlade client application with the IBM Informix GLS libraries.

For information about how to write and compile DataBlade client applications, see the *IBM Informix DataBlade API Programmer's Guide*.

You must specify the location of the Informix GLS header file, `ifxgls.h`. This header file is located in `%INFORMIXDIR%\incl\public` for Windows and in `$INFORMIXDIR/incl/public` for UNIX.

This directory also contains many of the other files that a DataBlade client application uses. Therefore, it is an important directory to include when you compile the application.

The following command uses the Microsoft compiler, cl, to compile a DataBlade client application called sample.c that uses Informix GLS:

```
cl -MD -Id:\msdev\include -Id:\informix\incl\public -Id:\informix\incl\esql \
-Id:\informix\incl -c sample.c
```

All Informix GLS functions are exported through the libthdmi.lib library. For linking, you must specify the following two libraries.

| At link time (in lib subdirectory of INFORMIXDIR) | At run time (in bin subdirectory of INFORMIXDIR) | Purpose |
|---|---|---|
| libthdmi.lib | idmit09a.dll | DataBlade API functions |
| isqlt09a.lib | isqlt09a.dll | ESQL/C functions |

Make sure to include the $INFORMIXDIR/incl/public directory with the -I compiler option, as follows:

```
cc -I$INFORMIXDIR/incl/public -I$INFORMIXDIR/esql -I$INFORMIXDIR/incl -c sample.c
```

The preceding command checks to see whether the API client libraries have been installed in the directory that the **INFORMIXDIR** environment variable indicates.

On UNIX platforms, you must explicitly specify the location and name of the library, as follows:

- For the library location, use the following -L compiler option:
  -L$INFORMIXDIR/lib/esql
- For the library name, include the **-lifgls** command-line option.

The following command links the sample.o object file to create an executable file called sample:

```
cc -o sample sample.o \
-L$INFORMIXDIR/lib/esql -L$INFORMIXDIR/lib/dmi -L$INFORMIXDIR/lib \
-lifdmi -lifsql -lifasf -lifcss -lifos -lifgen -lifgls -lifglx \
-lnsl -lsocket -laio -lm -ldl -lelf -lw \
$INFORMIXDIR/lib/esql/checkapi.o
```

Alternatively, you can use the ESQL/C preprocessor, **esql**, to compile and link the DataBlade client application. The **esql** preprocessor automatically links most of the libraries that the client application requires (including the Informix GLS library). However, you must explicitly specify the libdmi library. The following **esql** command compiles and links the sample.o object file:

```
esql -o sample sample.o -L$INFORMIXDIR/lib/dmi -ldmi
```

**Related concepts**:

"DataBlade user-defined routines"

**DataBlade user-defined routines:**

This topic provides information about how to compile and link a DataBlade user-defined routine (UDR).

For more information about how to write user-defined routines, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*, the *IBM Informix DataBlade API Programmer's Guide*, and the DataBlade Developers Kit documentation.

The GLS libraries are already linked to the database server. All IBM Informix GLS functions are exported through the SAPI.LIB library. Therefore, you do not need to specify the Informix GLS library explicitly when you compile and link a DataBlade UDR on Windows. For example, the following commands compile and link a DataBlade UDR called func1() that uses the Informix GLS library:

```
cl /DNT_MI_SAPI /DMI_SERVBUILD
-Id:\msdev\include -Id:\informix\incl\public -Id:\informix\incl
-c func1.c
link /DLL /OUT:func1.dll /DEF:func1.def func1.obj d:\informix\lib\SAPI.LIB
```

In the preceding example, d: specifies the drive on which Informix software is installed; informix is the directory that **INFORMIXDIR** specifies. The resulting dynamic link libraries (DLLs) must have the READONLY attribute set with the **attrib +r** command.

The Microsoft compiler, cl, defaults to the /MT option, which specifies the multithreaded library for DLL builds. This multithreaded library is statically linked to the DLL. You can specify the /MD option to put MSVCRT40.DLL in a command path. All the UDRs that are built with /MD use one copy of the LIBC DLL. This option prevents virtual-memory buildup in the database server when many UDRs are loaded.

The following example shows the use of the /MD option to compile the func1() DataBlade UDR:

```
cl /MD /DNT_MI_SAPI /DMI_SERVBUILD \
-Id:\msdev\include -Id:\informix\incl\public -Id:\informix\incl
-c func1.c
```

If MSVCRT40.DLL is not visible to the database server, the database server puts the following error in the online.log file when it attempts to execute the UDR:

```
Error loading UDR
```

On UNIX platforms, the GLS libraries are not linked to the database server. Therefore, you must explicitly specify the location and name of the Informix GLS library, as you do for DataBlade client applications.

The location of the Informix GLS header file, ifxgls.h, is the $INFORMIXDIR/incl/public directory. This directory contains many of the other files that a DataBlade client application uses. Make sure that you include this directory with the -I option of the compiler.

**Tip:** When you use the DataBlade Developers Kit to compile user-defined routines, you do not have to specify the location of the header files explicitly.

Additionally, you need to indicate that the DataBlade module is a DataBlade UDR (which runs on the server computer), not a DataBlade client application (which runs on a client computer). Include the MI_SERVBUILD compiler flag when you compile a DataBlade UDR.

The following example builds a shared object called udrs.so that contains the C code for a DataBlade UDR called func1():

```
% cc -KPIC -DMI_SERVBUILD -I$INFORMIXDIR/incl/public -I$INFORMIXDIR/incl
-L$INFORMIXDIR/esql/lib -c func1.c
% ld -G func1.o -o udrs.so
```

**Related concepts**:

"DataBlade client applications" on page 1-5

## Initialize the Informix GLS library

Before you can use IBM Informix GLS functions in your application, these functions must be able to access the current processing locale.

The **ifx_gl_init()** function initializes the current processing locale to the client locale. It creates a global locale structure that the Informix GLS functions can access to obtain the name of the current processing locale.

Whether an application requires a call to the **ifx_gl_init()** function depends on whether the application establishes its own connection to a database server or executes in the context of an existing database server connection:

- ESQL/C applications establish their own connections to a database server and therefore must call the **ifx_gl_init()** function at the beginning of the **main()** program block to establish the client locale as the current processing locale.
- DataBlade client applications establish their own connections to a database server and therefore must call the **ifx_gl_init()** function at the beginning of the **main()** program block to establish the client locale as the current processing locale.
- DataBlade user-defined routines (UDRs) execute in the context of an established connection and therefore use the server-processing locale as their current processing locale.

  The database server uses a server-processing locale to obtain locale information for its own internal sessions and for any connections. It uses the values of the client locale, database locale, server locale, and other environment information to determine the server-processing locale. DataBlade UDRs are not required to call the **ifx_gl_init()** function. For more information about the server-processing locale and how the database server establishes it, see the *IBM Informix GLS User's Guide*.

**Important:** You must ensure that the database server has established a current processing locale for an application before the application uses any other Informix GLS functions. If an application requires the **ifx_gl_init()** function, it must call **ifx_gl_init()** before it calls any other Informix GLS functions.

The **ifx_gl_init()** function does not generate an error if you call it multiple times within the same application. The first time that the application calls **ifx_gl_init()**, the function initializes the global locale structure. Subsequent calls do not result in multiple locale structures.

## Informix GLS exceptions

Many of the IBM Informix GLS functions return a unique value if they encounter an error. To further identify the cause of the error, you can use the **ifx_gl_lc_errno()** function to obtain the value of the Informix GLS error number.

The **ifx_gl_lc_errno()** function returns the error number as an int value. Informix GLS library functions can use **ifx_gl_lc_errno()** to obtain more information about an error that has occurred. The Informix GLS functions set the error number only if an error occurs unless a particular function is documented otherwise.

Because any Informix GLS library function can set the error number, you must call **ifx_gl_lc_errno()** immediately after the Informix GLS function in which the error occurred to inspect the relevant value of the error number.

For example, the **ifx_gl_mbslen()** function returns a value of -1 when it encounters an error. The following code fragment shows how to use the **ifx_gl_lc_errno()** function to obtain the error number:

```
if ( ifx_gl_mbslen(mbs, n) == -1 )
    switch ( ifx_gl_lc_errno() )
    ...
```

An application can also ignore the return value of an Informix GLS function and check the **ifx_gl_lc_errno()** function if the application sets **ifx_gl_lc_errno()** to 0 before it calls the function.

```
ifx_gl_lc_errno() = 0;
(void) ifx_gl_mbslen(mbs, n);
if ( ifx_gl_lc_errno() != 0 )
    switch ( ifx_gl_lc_errno() )
    ...
```

If an Informix GLS library function does not return a unique value to indicate that an error has occurred, you must use the preceding technique to obtain the error number. For example, the following code fragment tests whether the **ifx_gl_ismupper()** function has detected an invalid multibyte character:

```
ifx_gl_lc_errno() = 0;
value = ifx_gl_ismupper(mb, IFX_GL_NO_LIMIT);
if ( ifx_gl_lc_errno() != 0 )
    switch ( ifx_gl_lc_errno() )
    ...
```

You can set **ifx_gl_lc_errno()** to 0 and check it after a function call only if the current processing locale was initialized correctly with **ifx_gl_init()**. For example, the following code fragment of a client application results in a memory fault because the current processing locale is uninitialized:

```
int wrong_func()
{
    ifx_gl_lc_errno() = 0;    /* A memory fault will occur here. */
    (void) ifx_gl_init();
    if ( ifx_gl_lc_errno() != 0 )
        switch ( ifx_gl_lc_errno() )
        ...
```

In the preceding code fragment, the **ifx_gl_lc_errno()** function can write or read the error number only after a call to the **ifx_gl_init()** function has correctly initialized the current processing locale.

The following code fragment of a client application executes successfully because it correctly initializes the server-processing locale:

```
int right_func()
{
    (void) ifx_gl_init();
    ifx_gl_lc_errno() = 0;
    if ( ifx_gl_lc_errno() != 0 )
        switch ( ifx_gl_lc_errno() )
        ...
```

For multithreaded ESQL/C applications, the error-number value is stored as a field in the thread control block. Therefore, multithreaded ESQL/C applications can use the error number.

**Related reference**:

Appendix A, "List of Informix GLS error numbers," on page A-1

## Allocate memory

If an IBM Informix GLS function allocates memory, this memory is only allocated within the function itself and is freed before the function returns.

No Informix GLS library function allocates memory that remains after the function returns. Therefore, you must allocate any memory for data that an Informix GLS function needs or returns.

DataBlade UDRs can assume that the Informix GLS functions allocate memory with a PER_ROUTINE memory duration.

You must ensure that memory is allocated for any return value or argument that an Informix GLS function provides to the calling program.

**Tip:** You must also handle memory allocation for multibyte-character and wide-character strings that an Informix GLS functions use.

**Related reference**:

"Managing memory for strings and characters" on page 2-24

## Input and output streams

Character data that contains Asian characters must be correctly processed in all I/O for a graphical user interface, clipboard, character terminal, file, and network.

The following IBM Informix GLS functions process input and output multibyte character streams:

- The **ifx_gl_getmb()** function calls a function that you define to obtain the bytes that form one multibyte character from a specified location.
- The **ifx_gl_putmb()** function calls a function that you define to put the bytes that form one multibyte character in a specified location.

## Run the program

A client application (ESQL/C or DataBlade) that uses the IBM Informix GLS library must be able to access the following directories at run time.

| Contents | Windows directory | UNIX directory |
|---|---|---|
| Subdirectories for locale and code-set conversion files | `%INFORMIXDIR%\gls` | `$INFORMIXDIR/gls` |
| Static and shared GLS libraries | `%INFORMIXDIR%\lib\esql` | `$INFORMIXDIR/lib/esql` |

A DataBlade UDR executes in the context of the SQL statement that called it. The calling program must have access to the preceding directories.

# Improve program performance

Performance of an application that uses IBM Informix GLS is enhanced in the following ways:

- Optimization that all Informix GLS functions automatically perform
- Optional use of wide characters in the application

## Optimize Informix GLS

You can optimize the IBM Informix GLS functions by performing the following tasks:

- Evaluate the requested Informix GLS function.
- Determine if the requested function is appropriate for the data that you are trying to process.
- Execute the code that is appropriate for the type of data.

For example, if you use the **ifx_gl_mbsnext()** function to traverse data that is encoded in a single-byte code set, the function is reduced to a macro that advances the character byte by byte. The function does not execute code that parses multibyte sequences because it determines that the data is single byte.

Additionally, when collation is based on code-set order rather than locale-specific order, functions use a binary compare (such as **strcmp()**) instead of algorithms that examine collation weights.

**Related concepts**:

"Single-byte characters" on page 2-1

## Processing wide characters

You can choose how to structure your data to improve performance. IBM Informix GLS provides wide-character versions of most of its multibyte processing functions.

Typically, wide-character processing functions are faster than multibyte-character functions. However, wide characters require more space. Therefore, character data is generally stored and retrieved from the database in its multibyte form.

To use wide-character processing:

1. Convert the multibyte string to a wide-character string with the **ifx_gl_mbstowcs()** function.
2. Process the wide-character string.
3. Convert the wide-character string back to a multibyte string with the **ifx_gl_wcstombs()** function.

This technique is cost effective if the data that you process is traversed more than once.

**Related concepts**:

"Wide characters" on page 2-3

"Conversion between multibyte and wide characters" on page 2-4

# Chapter 2. Character processing

These topics describe the character-processing capabilities of the IBM Informix GLS library.

These topics describe the following topics that the Informix GLS library supports:

- Character types: single byte, multibyte, and wide
- Character operations such as character classification and code-set conversion
- String operations such as string traversal and concatenation
- Other operations on multibyte and wide characters

*Table 2-1. Character-processing tasks in descending order of importance.*

| Character-processing task | See |
|---|---|
| String traversal | "String traversal" on page 2-18 |
| String processing (concatenation, copying, string-length determination, character searching) | "String processing" on page 2-19 |
| Memory allocation | "Managing memory for strings and characters" on page 2-24 |
| Code-set conversion | "Code-set conversion" on page 2-13 |
| Character classification | "Character classification" on page 2-5 |
| Case conversion | "Case conversion" on page 2-8 |
| Character comparison and sorting | "Character/string comparison and sorting" on page 2-20 |

Data-formatting issues such as date/time conversion are also important areas in your application to internationalize.

**Related reference**:

Chapter 3, "Data formatting," on page 3-1

## Types of characters

A GLS locale supports a particular *code set*, which maps characters to unique bit patterns called *code points*.

A particular code set can contain the single-byte characters or multibyte characters.

For a general introduction to code sets, single-byte characters, and multibyte characters, see the *IBM Informix GLS User's Guide*.

Use of the IBM Informix GLS library helps to remove most assumptions about the type of character that your application handles.

### Single-byte characters

A single-byte character can hold code-point values 0 - 255.

A single-byte character can use 7 or 8 bits of a byte to represent a character, as follows:

- The 7-bit characters make up the ASCII code set.
  These characters contain code points in the range 0 - 127.
- The 8-bit characters contain code points in the range 128 - 255.
  Only software that is 8-bit clean can correctly interpret 8-bit characters.

English, European, and Middle Eastern code sets support at most 256 characters. Therefore, code sets that support these languages consist of single-byte characters.

When your application processes only single-byte characters, it 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.

The IBM Informix GLS functions and macros that handle multibyte characters are optimized for single-byte characters. Use of single-byte characters with these functions does not involve the full algorithms that multibyte-processing involves.

**Related reference**:

"Optimize Informix GLS" on page 1-11

## Multibyte characters

A multibyte character can hold code-point values greater than 255. One multibyte character can range 2 - 4 bytes in length.

Asian code sets are multibyte code sets; they contain both single-byte and multibyte characters.

If your application processes multibyte characters, it can no longer make the same assumption as for single-byte characters. The number of bytes in a buffer no longer 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 the following operations correctly:

- Allocate space for a multibyte-character string
- Traverse a multibyte-character string
- Find the beginning of the nth character in a multibyte-character string

Your application cannot use the built-in scaling of the C compiler for multibyte-character strings, but it can use the macros and functions of the IBM Informix GLS library to perform these operations on multibyte-character strings. To process a multibyte character, you cannot pass the entire character to a function. You must pass a pointer to the beginning of the character so that the called function can access the remaining bytes of the character.

For a list of operations that the functions of the Informix GLS library can perform on multibyte characters see "Character operations" on page 2-5. For a list of operations that the functions of the Informix GLS library can perform on multibyte-character strings, see "String operations" on page 2-18.

One single-byte assumption can still be applied to multibyte-character strings: no multibyte character has the null byte (0x000) as its second, third, or fourth byte. Therefore, if code is checking for only the single-byte ASCII null character, that code does not need to change to handle multibyte characters. This null character is also the null terminator in a multibyte character.

The names of most Informix GLS functions that handle multibyte characters start with one of the following strings:

**ifx_gl_mb**
Handles a multibyte character

**ifx_gl_mbs**
Handles a multibyte-character string

For example, **ifx_gl_mblen()** determines the length of a multibyte character, and **ifx_gl_mbslen()** determines the length of a multibyte-character string.

## The gl_mchar_t data type

The IBM Informix GLS library represents a multibyte character with the **gl_mchar_t** data type. The `gls.h` header file defines the **gl_mchar_t** data type, and the `ifxgls.h` header file includes `gls.h`. Therefore, you must include `ifxgls.h` in any file that uses the **gl_mchar_t** data type (or any Informix GLS function).

**Restriction:** The **gl_mchar_t** data type is an opaque structure. Do not access the individual bytes of a multibyte character directly.

Because any character in a multibyte-character string might contain up to 4 bytes, and **gl_mchar_t** refers to only one of those bytes, you usually declare a multibyte variable as a pointer to a **gl_mchar_t** data type. For example, the following declaration creates the **mb_string** variable as a pointer to a multibyte-character string:

```
#include <ifxgls.h>;
...
gl_mchar_t *mb_string;
```

The preceding declaration assumes that the application allocates memory for the **mb_string** multibyte string elsewhere.

You can also cast a C string to a multibyte-character string, as follows:

```
char *string; /* allocated & initialized elsewhere */
...
gl_mchar_t *mbs = (gl_mchar_t *) string;
```

**Related reference**:

"Multibyte-character-string allocation" on page 2-24

## Wide characters

The IBM Informix GLS functions and macros that handle multibyte characters use special multibyte-processing algorithms to determine the size of multibyte characters. However, the handling of these full multibyte-processing algorithms can be significant. Therefore, the Informix GLS library provides support for *wide characters* as an alternative form for the processing of multibyte characters. Wide characters allow you to rely on the C compiler built-in scaling instead of the multibyte-processing algorithms.

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. This size must be equal to or greater than the largest character that an operating system can support, and it must match the size of an integer data type that the C compiler can scale (such as short int, int, and long int).

The names of most Informix GLS functions that handle wide characters start with one of the following strings:

**ifx_gl_wc**
Handles a wide character

**ifx_gl_wcs**
Handles a wide-character string

For example, the function **ifx_gl_wctomb()** converts a wide character to a multibyte character, and **ifx_gl_wcslen()** determines the length of a wide-character string.

**Related tasks**:

"Processing wide characters" on page 1-11

**The gl_wchar_t data type:**

The IBM Informix GLS library represents a wide character with the **gl_wchar_t** data type.

The `gls.h` header file defines the **gl_wchar_t** data type, and the `ifxgls.h` header file includes `gls.h`. Therefore, you must include `ifxgls.h` in any file that uses the **gl_wchar_t** data type (or any Informix GLS function).

**Restriction:** The **gl_wchar_t** data type is an opaque structure. Do not access the individual bytes of a wide character directly.

The **gl_wchar_t** data type is a fixed-length structure. Therefore, you can declare a variable as a pointer to a **gl_wchar_t** structure or as a **gl_wchar_t** structure directly. For example, the following declarations create the **wc_string** variable as a pointer to a wide-character string and **wc_string2** as a single wide character:

```
#include <ifxgls.h>:
...
gl_wchar_t *wc_string;
gl_wchar_t wc_string2;
```

The declaration of **wc_string** assumes that the application allocates memory for this wide-character string elsewhere. The declaration of **wc_string2** allocates one wide character.

You can compare or assign a single-byte ASCII character or character constant to a single wide character, as in the following code fragment:

```
gl_wchar_t wc = 'a'; /* assigns an ASCII character */
if (wc=='a') /* compares with an ASCII character */
    ...
```

**Related reference**:

"Wide-character-string allocation" on page 2-25

**Conversion between multibyte and wide characters:**

To use wide characters, you convert multibyte characters to their wide character equivalents, process the characters, and convert the wide characters back to their multibyte equivalents.

The IBM Informix GLS library supports conversion between a multibyte form of a code set and its wide-character form. Unlike code-set conversion, the actual integral value of each character does not change in this conversion.

To change all character data to wide characters, you must first locate the character data and then find all the places where it is assigned and passed to functions. Informix GLS functions perform the following tasks to convert between multibyte and wide characters:

- To convert from a multibyte character to a wide character, the code point of the multibyte character is assigned a fixed number of bytes for all values, padded on the left with zeros:
  - The **ifx_gl_mbtowc()** function performs this conversion from multibyte character to wide character.
  - The **ifx_gl_mbstowcs()** function performs this conversion from multibyte-character string to wide-character string.
- To convert from a wide character to a multibyte character, the code point of the wide character is assigned the minimum number of bytes necessary to represent the value:
  - The **ifx_gl_wctomb()** function performs this conversion from wide character to multibyte character.
  - The **ifx_gl_wcstombs()** function performs this conversion from wide-character string to multibyte-character string.

**Related tasks**:

"Processing wide characters" on page 1-11

## Character operations

The IBM Informix GLS library supports character classification, case conversion, and code-set conversion operations on multibyte and wide characters.

**Related reference**:

Chapter 4, "Informix GLS functions," on page 4-1

### Character classification

A GLS locale groups the characters of a code set into *character classes*. Each class contains characters that have a related purpose.

The contents of a character class can be language specific. For example, the lower class contains all alphabetic lowercase characters in a code set. In the default locale, the default code set groups the English characters a through z into the lower class, but it also includes lowercase characters such as á, è, î, õ, and ü.

The default code set on UNIX platforms is ISO8859-1. The default code set for Windows environments is Microsoft 1252.

For more information about the default locale and the default code set, see the *IBM Informix GLS User's Guide*.

The LC_CTYPE category of a GLS locale file defines the following character classes.

| Character class | Contains |
|---|---|
| alpha | Alphabetic characters:<br>• Single-byte alphabetic characters a through z and A through Z<br>• Any single-byte non-English characters that the locale defines<br>• Any multibyte alphabetic or digit characters that the locale defines<br>This class includes characters in the lower and upper classes. |
| lower | Lowercase alphabetic characters:<br>• Single-byte alphabetic characters a through z<br>• Any single-byte non-English lowercase characters that the locale defines<br>• Any multibyte lowercase characters that the locale defines<br>No characters in this class are in the upper class. |
| upper | Uppercase alphabetic characters:<br>• Single-byte alphabetic characters A through Z<br>• Any single-byte non-English uppercase characters that the locale defines<br>• Any multibyte uppercase alphabetic characters that the locale defines<br>No characters in this class are in the lower class. |
| digit | Single-byte decimal digits 0 through 9 |
| xdigit | Hexadecimal digits:<br>• Single-byte numeric digits 0 through 9<br>• Single-byte representations of hexadecimal digits a through f and A through F<br>This class includes characters in the digit class. |
| alnum | All characters in both the alpha and digit classes. |
| blank | Horizontal white space:<br>• Single-byte horizontal-space characters:<br>" " (ASCII 0x020)<br>tab (ASCII 0x009)<br>• Any multibyte horizontal-space characters that the locale defines |

| Character class | Contains |
|---|---|
| space | Horizontal and vertical white space:<br>• Single-byte horizontal-space characters as defined in the blank class<br>• Single-byte vertical-space characters: new line, vertical tab, form feed, carriage return<br>• Any multibyte vertical-space characters that the locale defines<br>This class includes characters in the blank class. |
| cntrl | Control characters:<br>• Single-byte control characters: ASCII 0x000 to 0x01F<br>• Any other control characters that the locale defines |
| graph | Graphical characters are all characters that have visual representation. This class includes characters in the alpha, lower, upper, digit, xdigit, and punct classes. |
| punct | Punctuation:<br>• Single-byte punctuation characters:<br>! @ # $ % ^ & * ( ) - = + \ \| ‘ ~ [ ] { } ; : ‘ “ , . ? < ><br>• Any non-ASCII punctuation characters that the locale defines |
| print | All printable characters<br>This class includes characters in the alpha, lower, upper, digit, xdigit, graph, and punct classes. |

Your application must not assume which characters belong in a particular character class. For example, it must not contain code such as the following example to determine whether a character is lowercase:

```
if ( one_char >= 'a' && one_char <= 'z' )
```

Instead, use functions in the IBM Informix GLS library to identify the class of a particular character. The following table lists the GLS character classes and the Informix GLS functions that test for these classes for both multibyte and wide characters.

*Table 2-2. Informix GLS character-class functions*

| Character class | Multibyte-character function | Wide-character function |
|---|---|---|
| alnum (alpha or digit) | **ifx_gl_ismalnum()** | **ifx_gl_iswalnum()** |
| alpha | **ifx_gl_ismalpha()** | **ifx_gl_iswalpha()** |
| lower | **ifx_gl_ismlower()** | **ifx_gl_iswlower()** |
| upper | **ifx_gl_ismupper()** | **ifx_gl_iswupper()** |
| blank | **ifx_gl_ismblank()** | **ifx_gl_iswblank()** |
| space | **ifx_gl_ismspace()** | **ifx_gl_iswspace()** |
| digit | **ifx_gl_ismdigit()** | **ifx_gl_iswdigit()** |

*Table 2-2. Informix GLS character-class functions (continued)*

| Character class | Multibyte-character function | Wide-character function |
|---|---|---|
| xdigit | **ifx_gl_ismxdigit()** | **ifx_gl_iswxdigit()** |
| cntrl | **ifx_gl_ismcntrl()** | **ifx_gl_iswcntrl()** |
| graph | **ifx_gl_ismgraph()** | **ifx_gl_iswgraph()** |
| punct | **ifx_gl_ismpunct()** | **ifx_gl_iswpunct()** |
| print | **ifx_gl_ismprint()** | **ifx_gl_iswprint()** |

These Informix GLS functions check the LC_CTYPE category of the current locale to determine whether a specified character belongs to the respective character classification. The following code fragment uses the **ifx_gl_ismlower()** function to determine whether a multibyte character is lowercase:

```
if ( ifx_gl_ismlower(one_char, char_size)
```

The Informix GLS functions in Table 2-2 on page 2-7 do not return a unique value if they encounter an error. To detect an error, initialize the **ifx_gl_lc_errno()** error number to 0 before you call one of these functions, and then call **ifx_gl_lc_errno()** immediately after you call the function. For example, the following code fragment performs error checking for the **ifx_gl_ismlower()** function:

```
/* Initialize the error number */
ifx_gl_lc_errno() = 0;

/* Determine if 'mb' character is lowercase */
value = ifx_gl_ismlower(mb, mb_size);

/* If the error number has changed, ifx_gl_ismlower()has
* set it to indicate the cause of an error */
if ( ifx_gl_lc_errno() != 0 )
    /* Handle error */
else if ( value != 0 )
    /* Character 'mb' is in lower class */
else if ( value == 0 )
    /* Character 'mb' is NOT in lower class */
```

## Case conversion

In many languages, alphabetic characters have an uppercase and lowercase representation. Your application must not assume the case equivalent for a particular character.

For example, it must not contain code such as the following to obtain the uppercase equivalent of the character in `lower_char`:

```
upper_char = lower_char - 'a' + 'A';
```

The preceding line works for the English characters of the ASCII code set. However, it does not work for 8-bit characters, such as à and À.

To handle case conversion in your application, use functions in the IBM Informix GLS library to obtain the case equivalent of a particular character. The following table lists the case-conversion operations and the Informix GLS functions that perform them, both the multibyte functions and their wide-character equivalents.

*Table 2-3. Informix GLS case-conversion functions*

| Case-conversion operation | Multibyte-character function | Wide-character function |
|---|---|---|
| Obtain the lowercase equivalent of the source character | **ifx_gl_tomlower()** | **ifx_gl_towlower()** |
| Obtain the uppercase equivalent of the source character | **ifx_gl_tomupper()** | **ifx_gl_towupper()** |

The Informix GLS case-conversion functions check the LC_CTYPE category of the current locale to determine the case equivalent of a source character. If the case equivalent that you want exists, the functions return an integer value that is the alphabetic case equivalent of the source character. If no case equivalent exists, these functions return the source character.

The following code fragment uses the **ifx_gl_tomupper()** function to perform the case conversion of a multibyte character:

```
ret = ifx_gl_tomupper(upper_char, lower_char, char_sz)
```

## Case conversion for multibyte characters

The **ifx_gl_tomlower()** and **ifx_gl_tomupper()** functions require three arguments:

- The multibyte character or string to convert
- The destination buffer for the converted multibyte character or string
- The number of bytes to read to obtain a single multibyte character

For a multibyte-character string, the size of the case-converted string might not equal the size of the unconverted string. Therefore, to perform case conversion on multibyte characters, you must take the following special processing steps:

- Determine whether you need to allocate a separate destination buffer; if a destination buffer is needed, determine its size.
- Determine the number of bytes that the case-conversion process has read and written.

**Determine when to allocate a destination buffer:**

Whether you can perform case conversion of multibyte characters in place depends on whether the number of bytes written to the destination buffer is the same as the number of bytes read from the source.

You can determine when to allocate a destination buffer as follows:

- If the **ifx_gl_case_conv_outbuflen()** function determines that the size of the source string and its case-converted value are exactly equal, you can perform case conversion in place.
- If the size of the case-converted value of the source string is not the same as the size of the source string itself, you cannot perform case conversion in place.

If you cannot perform case conversion in place, you must allocate a separate destination buffer. To allocate this buffer, you need to have an estimate of the number of bytes that it needs to hold. Use any of the following methods to determine the number of bytes that might be written to the destination buffer:

- The **ifx_gl_case_conv_outbuflen()** function calculates either exactly the number of bytes that will be written to the destination buffer or a close over approximation of the number.

  This function applies to both uppercase and lowercase conversions. The second argument to **ifx_gl_case_conv_outbuflen()** is the number of bytes in the character source.
- The **ifx_gl_mb_loc_max()** function calculates the maximum number of bytes that can be written to the destination buffer for any source value in the current locale.

  This value is always greater than or equal to (>=) the value that the **ifx_gl_case_conv_outbuflen()** function returns.
- The macro IFX_GL_MB_MAX returns the maximum number of bytes that can be written to the destination buffer for any source value in any locale.

  This value is always greater than or equal to the value that the **ifx_gl_mb_loc_max()** function returns.

Of the preceding options, the macro IFX_GL_MB_MAX is the fastest and the only method that can initialize static buffers. The function **ifx_gl_case_conv_outbuflen()** is the slowest but the most precise.

The following code fragment uses the **ifx_gl_mblen()** function to determine the size of the source character and the **ifx_gl_case_conv_outbuflen()** function to determine the estimated size of the case-converted value:

```
/* Obtain the sizes of the source and destination strings */
src_mb_bytes = ifx_gl_mblen(src_mb, ...);
dst_mb_bytes = ifx_gl_case_conv_outbuflen(src_mb_bytes);

if ( dst_mb_bytes == src_mb_bytes )
/* Sizes of source and case-equivalent characters are the
* same. Perform the case conversion in place */
   {
   retval =
       ifx_gl_tomupper(src_mb, src_mb, IFX_GL_NO_LIMIT);
   }
else
/* Sizes of source and destination characters are NOT the
 * same. Allocate a destination buffer and perform case
 * conversion into this buffer */
   {
   dst_mb = (gl_mchar_t *) malloc(dst_mb_bytes);
   retval =
       ifx_gl_tomupper(dst_mb, src_mb, IFX_GL_NO_LIMIT);
   }
```

**Related concepts**:

"Multibyte-character termination" on page 2-23

**Determine the number of bytes read and written:**

The **ifx_gl_tomupper()** and **ifx_gl_tomlower()** functions return an unsigned short integer that encodes the information about the number of bytes that the function has read.

The Informix GLS library provides the following macros to obtain this information from the return value.

**IFX_GL_CASE_CONV_SRC_BYTES()**
: The number of bytes read from the source string

**IFX_GL_CASE_CONV_DST_BYTES()**
The number of bytes written to the destination buffer

The following code fragment uses the **ifx_gl_tomlower()** function to convert a multibyte character to its lowercase equivalent. It uses the case-conversion macros to obtain the number of bytes read and written during the case-conversion operation:

```
/* Initialize source pointer, 'src_mb', to beginning of the
 * multibyte string. Initialize destination pointer to
 * beginning of destination buffer */
src_mb = src_mbs;
dst_mb = dst_mbs;

/* Traverse source string until the null terminator is
* reached */
while ( *src_mb != '\0' )
    {
/* Convert source multibyte character, 'src_mb', to lowercase
 * and put in the destination buffer */
    unsigned short retval =
        ifx_gl_tomlower(dst_mb, src_mb, src_mbs_bytes);
...
/* Increment the source pointer by the number of bytes that
 * have been read and the destination pointer by the number
 * of bytes that have been written */
    src_mb += IFX_GL_CASE_CONV_SRC_BYTES(retval);
    dst_mb += IFX_GL_CASE_CONV_DST_BYTES(retval);
    src_mbs_bytes -= IFX_GL_CASE_CONV_SRC_BYTES(retval);
    }
```

The memory-management rules for case conversion of a single multibyte character also apply to converting a string of one or more multibyte characters. For example, the following code fragment converts a multibyte-character string to its uppercase equivalent:

```
/* Assume src_mbs is null terminated */
src_mbs_bytes = strlen(src_mbs);
dst_mbs_bytes = ifx_gl_case_conv_outbuflen(src_mbs_bytes);

if ( dst_mbs_bytes == src_mbs_bytes )
    {
    /* If two strings have the same size, overwrite each
     * multibyte character in the 'src_mbs' multibyte string
     * with its uppercase equivalent */
    src_mb = src_mbs;
    while ( *src_mb != '\0' )
        {
        retval =
           ifx_gl_tomupper(src_mb,src_mb,IFX_GL_NO_LIMIT);
        src_mb += IFX_GL_CASE_CONV_SRC_BYTES(retval);
        }
    }
else
    {
    /* Two strings are not the same size, so must allocate a
     * destination buffer whose size is determined by the
     * ifx_gl_case_conv_outbuflen() function */
    dst_mbs = (gl_mchar_t *) malloc(dst_mbs_bytes + 1);

    src_mb = src_mbs;
    dst_mb = dst_mbs;

    while ( *src_mb != '\0' )
        {
        retval =
```

```
            ifx_gl_tomupper(dst_mb,src_mb,IFX_GL_NO_LIMIT);
        src_mb += IFX_GL_CASE_CONV_SRC_BYTES(retval);
        dst_mb += IFX_GL_CASE_CONV_DST_BYTES(retval);
        }

    *dst_mb = '\0';
    }
```

## Case conversion for wide characters

Because a wide character has a fixed size, the **ifx_gl_towlower()** and **ifx_gl_towupper()** functions require only one argument: the wide character to convert.

These functions return an integer value of the case-equivalent character for this wide character. Therefore, you can always perform case conversion of wide characters in place. For example, you can assign the case equivalent of **src_wc** back to **src_wc**, as follows:

```
src_wc = ifx_gl_towupper(src_wc);
```

You can also perform case conversion of wide characters to a destination buffer. The previous line could also be written as follows:

```
dst_wc = ifx_gl_towupper(src_wc);
```

## Exception handling

These case-conversion functions do not return a special value if they encounter an error. To detect an error, initialize the **ifx_gl_lc_errno()** error number to 0 before you call one of these functions and check **ifx_gl_lc_errno()** immediately after you call it. The following code fragment performs exception handling in the conversion of a wide character to its lowercase equivalent:

```
/* Initialize the error number */
ifx_gl_lc_errno() = 0;

/* Perform conversion of 'src_wc' to lowercase */
dst_wc = ifx_gl_towlower(src_wc);

/* If the error number has changed, ifx_gl_towlower() has set
 * it to indicate the cause of an error */
if ( ifx_gl_lc_errno() != 0 )
    /* Handle error */
else
    ...
```

## Performance issues

The IBM Informix GLS case-conversion functions assign the destination character regardless of whether the source character has a case-equivalent character. If no case equivalent for a particular source character exists, the functions return only the source character. Therefore, the following two algorithms perform the same task:

- Calling the case-conversion function regardless of the existence of a case-equivalent character, as follows:

  ```
  dst_wc = ifx_gl_towlower(src_wc);
  ```

- Calling the case-conversion function only if a case-equivalent character exists, as follows:

  ```
  if ( ifx_gl_iswupper(src_wc) )
     dst_wc = ifx_gl_towlower(src_wc);
  else
     dst_wc = src_wc;
  ```

However, the first approach is usually faster.

## Code-set conversion

A character might be encoded differently on two different operating systems. Therefore, the appropriate communication layer must be prepared to convert between the two encodings.

This process of conversion between two code sets is called *code-set conversion*. Code-set conversion translates code points from a source code set to a destination code set.

IBM Informix ESQL/C applications automatically perform any needed code-set conversion when they send and receive database data.

The following principles apply when you send and receive database data:

- DataBlade client applications automatically perform any needed code-set conversion.
- DataBlade UDRs do not automatically perform code-set conversion.

For an introduction to code-set conversion, see the *IBM Informix GLS User's Guide*.

If your application needs to perform code-set conversion, it must:

- Determine if code-set conversion is needed between the source and target code sets.
- Perform the code-set conversion on a character string if it is needed.

### Determining if code-set conversion is needed

The **ifx_gl_conv_needed()** function determines whether characters encoded in a source code set require conversion to a destination code set. Use this function to determine if code-set conversion is needed.

Comparing the names of the code sets does not provide enough information to determine if it is necessary. In the **ifx_gl_conv_needed()** function, you can specify the source and destination code sets as any of the following items:

- Locale names
- Code-set names
- The IFX_GL_PROC_CS macro

**Related concepts**:

"Specify code-set names" on page 2-14

### Perform code-set conversion

For a multibyte-character string, the size of the converted string might not equal the size of the unconverted string. Therefore, to perform code-set conversion on multibyte characters, you must take the following special processing steps:

- Determine whether you need to allocate a separate destination buffer; if a destination buffer is needed, determine its size.
- Preserve state information for multiple fragments of a multibyte string.

**Determine when to allocate a destination buffer:**

Whether you can perform code-set conversion on multibyte characters in place depends on whether the number of bytes written to the destination buffer is the same as the number of bytes read from the source.

Determine when to allocate a destination buffer as follows:

- If the **ifx_gl_cv_outbuflen()** function determines that the size of the source string and its code-set-converted value are exactly equal, you can perform code-set conversion in place.
- If the size of the code-set-converted value of the source string is not the same as the size of the source string itself, you cannot perform code-set conversion in place.

If you cannot perform code-set conversion in place, you must allocate a separate destination buffer. To allocate a destination buffer, you need to have an estimate of the number of bytes that it needs to hold. You can use either of the following methods to determine the number of bytes that might be written to the destination buffer:

- The **ifx_gl_cv_outbuflen()** function calculates either exactly the number of bytes that will be written to the destination buffer or a close over-approximation of the number.

  The third argument to **ifx_gl_cv_outbuflen()** is the number of bytes in the character source.
- The IFX_GL_MB_MAX macro can be used in the following expression to calculate the maximum number of bytes that can be written to the destination buffer for any source value in any locale:

  *src_bytesleft* * IFX_GL_MB_MAX

  The *src_bytesleft* value is the number of bytes to convert. This expression value is always greater than or equal to the expression value that uses the **ifx_gl_mb_loc_max()** function.

Of the two options, the expression that uses the macro IFX_GL_MB_MAX is faster and can be used to initialize static buffers. The function **ifx_gl_case_conv_outbuflen()** is slower but more precise.

The following code fragment uses the **ifx_gl_cv_outbuflen()** function to determine the estimated size of a code-set-conversion destination buffer:

```
int dstbytes;
gl_mchar_t *dstmbs;
conv_state_t state;

dstbytes = ifx_gl_cv_outbuflen("ujis", "sjis", srcbytes);
dstmbs = (gl_mchar_t *) malloc(dstbytes);

state.first_frag = 1;
state.last_frag = 1;
if (ifx_gl_cv_mconv(&state, &dstmbs, &dstbytes, "ujis"
    &srcmbs, &srcbytes, "sjis") == -1 )
```

**Related concepts**:

"Preserve state information" on page 2-15

**Specify code-set names:**

You can specify the names of the source and destination code sets with either locale names, code-set names, or the IFX_GL_PROC_CS macro.

- Locale names

For example, you can use de_de.8859-1 for the German locale or ja_jp.ujis for the Japanese UJIS locale. For more information about locale names, see the *IBM Informix GLS User's Guide*.

- Code-set names

  You can find the names of code sets in code-set name registry.

  The code-set name registry is in `%INFORMIXDIR%\gls\cmZ` for Windows and in `$INFORMIXDIR/gls/cmZ` for UNIX.

  In the preceding path names, **INFORMIXDIR** is the environment variable that specifies the directory where you install the IBM Informix product, and Z represents the version number for the code-set object-file format.

- The IFX_GL_PROC_CS macro

  This macro specifies use of the code set of the current processing locale. Depending on the context, the value of IFX_GL_PROC_CS is based on either the client environment or the database that the database server is currently accessing.

The preceding formats are valid as code-set names in any of the following Informix GLS functions:

- **ifx_gl_conv_needed()**
- **ifx_gl_cv_mconv()**
- **ifx_gl_cv_outbuflen()**
- **ifx_gl_cv_sb2sb_table()**

**Related concepts**:

"Determining if code-set conversion is needed" on page 2-13

**Preserve state information:**

Most code sets are not state dependent; that is, the characters of these code sets can be decoded with only one algorithm, and each byte sequence represents a unique character.

In contrast, byte sequences in state-dependent code sets can represent more than one character. Which character a sequence represents depends on the current state. State-dependent code sets occur primarily on IBM mainframe computers, and they affect only code-set conversion.

When you fragment a complete source string into two or more nonadjacent source buffers, you must call the **ifx_gl_cv_mconv()** function multiple times, to perform code-set conversion on each fragment of the string. Because of the nature of state-dependent code sets (and because the caller of this function cannot know whether either the source or destination code set is a state-dependent code set), you must preserve state information across the multiple calls of **ifx_gl_cv_mconv()**. The **ifx_gl_cv_mconv()** argument state is used for this purpose.

The *state* argument is a pointer to a **conv_state_t** structure. This structure contains two fields that you must set to indicate that you are performing code-set conversion on fragmented strings: **first_frag** and **last_frag**. The following table lists the different fragments of a string and the corresponding values to which you must set these two **conv_state_t** fields.

| String fragment | Value of first_frag field | Value of last_frag field |
|---|---|---|
| String is the first of n fragments. | 1 | 0 |

| String fragment | Value of first_frag field | Value of last_frag field |
|---|---|---|
| String is the 2nd, ..., nth-1 fragment. | 0 | 0 |
| String is the last (nth) fragment. | 0 | 1 |
| String is not fragmented; it is a complete string. | 1 | 1 |

**Important:** The **conv_state_t** structure contains other fields that are for internal use only. IBM Informix does not guarantee that these other internal fields of **conv_state_t** will not change in future releases. Therefore, to create portable code, set only the **first_frag** and **last_frag** fields of the **conv_state_t** structure.

Pass the fragments to the **ifx_gl_cv_mconv()** function in the same order in which they appear in the complete string. Use the same **conv_state_t** structure for all of the fragments of the same complete string.

The following code performs code-set conversion on a complete character string that is not fragmented:

```
int unfrag_strng(out_str, out_len, out_cs, in_str,
        in_len, in_cs)
    gl_mchar_t *out_str;
    int out_len;
    char *out_cs;
    gl_mchar_t *in_str;
    int in_len;
    char *in_cs;
{
    conv_state_t state;
    int ret;

    state.first_frag = 1;
    state.last_frag = 1;
    ret = ifx_gl_cv_mconv(&state, &out_str, &out_len,
            out_cs, &in_str, &in_len, in_cs);
    ...
}
```

This code assigns both the **first_frag** and **last_frag** fields a value of 1 to indicate that the multibyte string is not fragmented.

Suppose that you have a complete multibyte-character string that is fragmented into four different buffers. The following code performs code-set conversion on this fragmented string:

```
int frag_strng(out_str, out_len, out_cs, in_str,
        in_len, in_cs)
    gl_mchar_t *out_str;
    int out_len;
    char *out_cs;
    gl_mchar_t *in_str[];
    int in_len;
    char *in_cs;
{
    conv_state_t state;
    int ret;
/* Perform code-set conversion on the FIRST fragment:
 * first_frag = 1; last_frag = 0 */
   state.first_frag = 1;
   state.last_frag = 0;
```

```
   ret = ifx_gl_cv_mconv(&state, &out_str, &out_len, out_cs,
       &in_str[0], &in_len, in_cs);
   ...
/* Perform code-set conversion on the SECOND fragment:
    first_frag = 0; last_frag = 0 */
    state.first_frag = 0;
    state.last_frag = 0;
    ret = ifx_gl_cv_mconv(&state, &out_str, &out_len, out_cs,
        &in_str[1], &in_len, in_cs);
    ...
/* Perform code-set conversion on the THIRD fragment.
 * No need to set the first_frag and last_frag fields again,
 * because they are already 0 */
    ret = ifx_gl_cv_mconv(&state, &out_str, &out_len, out_cs,
        &in_str[2], &in_len, in_cs);
    ...
/* Perform code-set conversion on the FOURTH (last)
 * fragment: first_frag = 0; last_frag = 1 */
    state.first_frag = 0;
    state.last_frag = 1;
    ret = ifx_gl_cv_mconv(&state, &out_str, &out_len, out_cs,
        &in_str[3], &in_len, in_cs);
    ...
}
```

**Related concepts**:

"Determine when to allocate a destination buffer" on page 2-13

"Fragment multibyte strings" on page 2-27

## Performance issues

Most performance overhead in code-set conversion is a result of either memory management or multibyte-string traversal. However, only if one of the code sets is a multibyte code set does code-set conversion require this overhead to convert correctly. If the code-set conversion is between two single-byte code sets, you can obtain a code-set conversion table and avoid this overhead.

The following sample code uses the **ifx_gl_cv_sb2sb_table()** function to obtain a code-set conversion table for two single-byte code sets:

```
void do_codeset_conversion(src, src_codeset, dst,
        dst_codeset)
    unsigned char *src;
    char *src_codeset;
    unsigned char *dst;
    char *dst_codeset;
{
    unsigned char *table;

    if ( ifx_gl_cv_sb2sb_table(dst_codeset,
            src_codeset, &table) == -1 )
      /* Handle Error */

    if ( table != NULL )
        {
        /* Convert in place */
        for ( ; *src != '\0'; src++ ) *src = table[*src];
            dst = src;
        }
    else
       {
       /* Full GLS code-set conversion, which handles
        * multibyte conversions and complex conversions
```

```
        * between single-byte code sets */
       ...
       }
}
```

# String operations

The IBM Informix GLS library supports string traversal and string processing operations on multibyte-character and wide-character strings.

**Related reference**:

Chapter 4, "Informix GLS functions," on page 4-1

## String traversal

Because a single-byte character occupies only one byte, string traversal of a single-byte character string can use the built-in scaling of the C compiler. However, you must take special steps to handle string traversal of multibyte-character and wide-character strings.

### Multibyte-character-string traversal

Because a multibyte-character string might contain multibyte characters of different sizes, you cannot automatically traverse the string with any built-in scaling. Instead, use the following functions in IBM Informix GLS to traverse a multibyte string:

- The **ifx_gl_mblen()** function determines the number of bytes in a multibyte character.
- The **ifx_gl_mbsnext()** function returns a pointer to the next multibyte character in the multibyte string.
- The **ifx_gl_mbsprev()** function returns a pointer to the previous multibyte character in the multibyte string.

These functions support string traversal in the following directions:

- Forward

```
gl_mchar_t buf[], *p;
for ( p = buf; *p != '\0' ;
       p = ifx_gl_mbsnext(p, IFX_GL_NO_LIMIT))
   process_mchar(p);
```

- Backward

```
gl_mchar_t buf[], *p;
p = ifx_gl_mbsprev(buf, buf + strlen(buf));
if ( p != NULL )
   for ( ; p >= buf; p = ifx_gl_mbsprev(buf, p) )
      process_mchar(p);
```

### Wide-character-string traversal

A wide-character string consists of fixed-length characters. The IBM Informix GLS library does not need to provide traversal functions for wide-character strings because you can traverse the string with the built-in scaling of the C compiler. However, the library does provide the **ifx_gl_wcslen()** function, which determines the number of bytes in a wide-character string.

The Informix GLS library supports wide-character string traversal in the following directions:

- Forward

```
gl_wchar_t buf[], *p;
for ( p = buf; *p != '\0' ; p++ )
   process_wchar(*p);
```

- Backward

```
gl_wchar_t buf[], *p;
for ( p = buf + ifx_gl_wcslen(buf) - 1; p >= buf; p-- )
   process_wchar(*p);
```

# String processing

The IBM Informix GLS library provides functions for concatenation, string copying, string-length determination, and character searching. The Informix GLS library provides both multibyte and wide-character functions for these string-processing tasks.

## Concatenation

*Concatenation* is the process of appending one string to the end of another string.

The following table lists the IBM Informix GLS multibyte functions and their wide-character equivalents that perform string concatenation.

| Concatenation operation | Multibyte-character function | Wide-character function |
|---|---|---|
| Append a copy of one character string to the end of another character string. If the two strings overlap, the result of the operation is undefined. | **ifx_gl_mbscat()** | **ifx_gl_wcscat()** |
| Append a specified number of characters from one character string to the end of a second character string. If the two strings overlap, the result of this operation is undefined. | **ifx_gl_mbsncat()** | **ifx_gl_wcsncat()** |

## String copying

The following table lists the IBM Informix GLS multibyte functions and their wide-character equivalents that perform string copying.

| String-processing task | Multibyte-character function | Wide-character function |
|---|---|---|
| Copy one character string to a specified location in a second character string. If the two strings overlap, the result of this operation is undefined. | **ifx_gl_mbscpy()** | **ifx_gl_wcscpy()** |
| Copy a specified number of characters from one character string to a second character string. If the two strings overlap, the result of this operation is undefined. | **ifx_gl_mbsncpy()** | **ifx_gl_wcsncpy()** |

### String-length determination

The following table lists the IBM Informix GLS multibyte functions and their wide-character equivalents that determine string length or number of characters.

| String-length task | Multibyte-character function | Wide-character function |
|---|---|---|
| Determine the number of characters in the string, not including any terminating null characters. | **ifx_gl_mbslen()** | **ifx_gl_wcslen()** |
| Determine the number of bytes in a multibyte string, not including any trailing space characters. | **ifx_gl_mbsntsbytes()** | None |
| Determine the number of characters in a string, not including the trailing space characters. | **ifx_gl_mbsntslen()** | **ifx_gl_wcsntslen()** |
| Determine the number of characters in the initial substring of one string that consists entirely of characters in a second specified character string. | **ifx_gl_mbsspn()** | **ifx_gl_wcsspn()** |
| Determine the number of characters in the initial substring of one string that consists entirely of characters not in a specified second character string. | **ifx_gl_mbscspn()** | **ifx_gl_wcscspn()** |

### Character searching

The following table lists the IBM Informix GLS multibyte functions and their wide-character equivalents that perform character searching.

| Character-searching task | Multibyte-character function | Wide-character function |
|---|---|---|
| Search for the first occurrence of a character in the character string. | **ifx_gl_mbschr()** | **ifx_gl_wcschr()** |
| Search for the last occurrence of a character in the character string. | **ifx_gl_mbsrchr()** | **ifx_gl_wcsrchr()** |
| Search for a substring in another string. | **ifx_gl_mbsmbs()** | **ifx_gl_wcswcs()** |
| Search for the first occurrence in the character string of any character from a second specified character string. | **ifx_gl_mbspbrk()** | **ifx_gl_wcspbrk()** |

## Character/string comparison and sorting

Collation involves the sorting of character data that is either stored in a database or manipulated in a client application.

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

- Code-set collation order is the bit-pattern order of characters within a code set.
  The order of the code points in the code set determines the sort order.
- Locale-specific order is an order of the characters that relates to a real language.
  The LC_COLLATE category of a GLS locale file defines the order of the characters in the locale-specific order.

For more information about code-set and locale-specific order, see the *IBM Informix GLS User's Guide*.

To perform code-set collation, you compare only the integer values of two multibyte or two wide characters. For example, suppose one multibyte character, **mbs1**, contains the value $A^1A^2A^3$ and a second multibyte character, **mbs2**, contains the value $B^1B^2B^3$. If the integer value of $A^1A^2A^3$ is less than the integer value of $B^1B^2B^3$, then **mbs1** is less than **mbs2** in code-set collation order.

However, sometimes you want to sort character data according to the language usage of the characters. In code-set order, the character `a` is greater than the character `A`. In many contexts, you would probably not want the string `Apple` to sort before the string `apple`. The locale-specific order could list the character `A` after the character `a`. Similarly, even though the character `À` might have a code point of 133, the locale-specific 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 following table lists the Informix GLS functions that use locale-specific order to compare two multibyte-character or wide-character strings.

| String-comparison task | Multibyte-character function | Wide-character function |
|---|---|---|
| Compare two character strings by locale-specific order. | **ifx_gl_mbscoll()** | **ifx_gl_wcscoll()** |

These functions access the LC_COLLATE category of a locale file to obtain localized collating information when they compare character strings. They return an integer value that indicates the results of the comparison between two string arguments. The following table shows the comparison between the first string argument (Arg 1) and the second string argument (Arg 2), as well as the return values.

| Argument comparison | Return value |
|---|---|
| Arg 1 < Arg 2 | <0 |
| Arg 1 = Arg 2 | 0 |
| Arg 1 > Arg 2 | >0 |

The **ifx_gl_mbscoll()** and **ifx_gl_wcscoll()** functions do not return a special value if an error has occurred. Therefore, to detect an error condition, you must initialize the **ifx_gl_lc_errno()** error number to zero before you call one of these functions and check **ifx_gl_lc_errno()** after you call the function. The following code fragment performs error checking for a call to the **ifx_gl_wcscoll()** function:

```
/* Initialize the error number */
ifx_gl_lc_errno() = 0;
```

```
/* Compare the two wide-character strings */
value = ifx_gl_wcscoll(wcs1, wcs1_char_length, wcs2,
wcs2_char_length);

/* If the error number has changed, ifx_gl_wcscoll() has
 * set it to indicate the cause of an error */
if ( ifx_gl_lc_errno() != 0 )
    /* Handle error */
else if ( value < 0 )
    /* wcs1 is less than wcs2 */
else if ( value == 0 )
    /* wcs1 is equal to wcs2 */
else if ( value > 0 )
    /* wcs1 is greater than wcs2 */
...
```

# Other operations

In addition to the operations on characters and strings, the IBM Informix GLS library provides support for terminating characters and strings, allocating memory for characters and strings, and truncating and fragmenting strings.

## String and character termination

You can use the IBM Informix GLS library with many different application programming interfaces (APIs), which might handle strings in different ways.

To provide flexible support for APIs, the Informix GLS library allows you to indicate how to handle the following items:

- Character-string termination

  Is the string argument a null-terminated string?
- Multibyte-character termination

  Is the length of a multibyte character known?

### Character-string termination

The API that you use with the IBM Informix GLS library might handle string termination in either of the following ways:

- All character strings are terminated with a null character.

  The null character indicates the end of the string. Such strings are called null-terminated strings. The null terminator of a multibyte string consists of one byte whose value is 0. The null terminator of a wide-character string consists of one **gl_wchar_t** character whose value is 0.
- Each string consists of a pointer and length that indicates the number of bytes in the string.

  Character strings that are not null-terminated are called length-terminated strings. Length-terminated strings can contain null characters, but these null characters do not indicate the end of the string.

The Informix GLS functions that take a string argument allow you to pass this string as either a null-terminated string or a length-terminated string. To provide this flexibility, many Informix GLS functions that accept a multibyte or wide-character string expect the string to be represented with the following two arguments:

- The string itself
- The length of the string

The value that you provide for the string length tells the Informix GLS function how to handle the associated string, as the following table shows.

| String-length value | Meaning |
|---|---|
| IFX_GL_NULL | The Informix GLS function assumes that the string is a null-terminated string. |
| >=0 | The Informix GLS function assumes that this length indicates the number of bytes in the length-terminated string. |
| <0, != IFX_GL_NULL | The Informix GLS function sets the error number to the IFX_GL_PARAMERR error. |

## Multibyte-character termination

Many GLS library functions operate on just one multibyte character. Each IBM Informix GLS function that accepts a multibyte character expects the character to be represented by the following two arguments:

- The character itself
- The maximum length of the character

The value that you provide for the character length tells the Informix GLS function how to handle the associated character, as the following table shows.

| String-length value | Meaning |
|---|---|
| IFX_GL_NO_LIMIT | The Informix GLS function reads as many bytes as necessary from the multibyte character to form a complete character. |
| >=0 | The Informix GLS function does not read more than this number of bytes from the multibyte character when it tries to form a complete character. |
| <0, != IFX_GL_NO_LIMIT | The Informix GLS function sets the error number to the IFX_GL_EINVAL error. |

If the multibyte character is in a null-terminated multibyte string, the character length must be IFX_GL_NO_LIMIT. For example, if mbs points to a null-terminated string of multibyte characters, the following code fragment must specify IFX_GL_NO_LIMIT as the character length:

```
for ( mb = mbs; *mb != '\0'; mb += bytes )
    {
    if ( (bytes = ifx_gl_mblen(mb, IFX_GL_NO_LIMIT)) == -1 )
        /* handle error */
    }
```

If a multibyte character, **mb**, is in a length-terminated multibyte string or is a character in a buffer by itself, the character length must equal the number of bytes between where **mb** points and the end of the buffer that holds the string or character. For example, if **mbs** points to a length-terminated string of multibyte characters and **mbs_bytes** is the number of bytes in that string, the following call to **ifx_gl_mblen()** must specify the length of the multibyte string:

```
for ( mb = mbs; mbs_bytes > 0; mb += bytes,
        mbs_bytes -= bytes )
    {
    if ( (bytes = ifx_gl_mblen(mb, mbs_bytes)) == -1 )
        /* handle error */
    }
```

Similarly, if **mb** points to one multibyte character and **mb_bytes** is the number of bytes in the buffer that holds the character, the following call to **ifx_gl_mblen()** must specify the length of the multibyte character:

```
if ( (bytes = ifx_gl_mblen(mb, mb_bytes)) == -1 )
    /* handle error */
```

If the Informix GLS function cannot determine whether bytes in a buffer make up a valid multibyte character, it sets the error number to IFX_GL_EINVAL. The function is unable to determine a valid multibyte character for the following reasons:

- The function would need to read more than the specified number of bytes to recognize a multibyte character.
- The specified character length is less than or equal to zero (<=0).

**Tip:** Wide characters are fixed length. Therefore, Informix GLS functions that operate on wide characters do not require the character length.

**Related concepts**:

"Determine when to allocate a destination buffer" on page 2-9

## Managing memory for strings and characters

You must make buffers large enough to hold text in any of the languages that your application will handle.

If your application will handle many languages, you must ensure that allocated buffers are large enough to hold translated versions of the text. If your application will handle Asian (multibyte) languages, you need to replace single-byte buffers with multibyte- or wide-character buffers.

**Important:** Any memory that IBM Informix GLS functions allocate remains allocated only for the duration of the function. It does not remain after the function returns. Therefore, you must manage memory for multibyte-character and wide-character strings.

**Related concepts**:

"Allocate memory" on page 1-10

### Multibyte-character-string allocation

Multibyte characters have varying lengths. When you represent a multibyte-character string in an array, the number of array elements does not equal the number of multibyte characters in the string. Therefore, you cannot use the same allocation method for multibyte strings as for single-byte strings.

Instead, you can use the following IBM Informix GLS macro and functions to help you determine how much memory a multibyte character requires.

**IFX_GL_MB_MAX**
: Indicates the maximum number of bytes that any multibyte character in any locale can occupy.

Use this macro to allocate space in static buffers that are intended to contain one or more multibyte characters.

**ifx_gl_mb_loc_max()**
Returns the maximum number of bytes that any character in the current locale can occupy.

**ifx_gl_cv_outbuflen()**
**ifx_gl_case_conv_outbuflen()**
Calculates one of the following values:

- Exactly the number of bytes that are required by a destination buffer of the converted multibyte characters
- A close over-approximation of the number

For example, the following declaration statically allocates 20 multibyte characters for the **mbs** string:

```
gl_mchar_t mbs[20 * IFX_GL_MB_MAX]; /* static allocation */
```

The following declarations dynamically allocate 20 multibyte characters for the **mb1** and **mb2** strings:

```
gl_mchar_t *mbs1 = (gl_mchar_t *) malloc(20*IFX_GL_MB_MAX);
gl_mchar_t *mbs2 = (gl_mchar_t *) malloc(20*ifx_gl_mb_loc_max());
```

The declaration for **mb1** uses the maximum multibyte-character size. The declaration for **mb2** uses the **ifx_gl_mb_loc_max()** function to obtain a more precise estimate for the size of 20 multibyte characters. The **ifx_gl_mb_loc_max()** function returns the maximum size among all characters in the current processing locale.

If your multibyte-character string is null terminated, allocate one additional byte for the null terminator. The following declarations allocate three null-terminated multibyte-character strings:

```
/* static allocation */
gl_mchar_t mbs[20*IFX_GL_MB_MAX+1];

/* dynamic allocation with IFX_GL_MB_MAX */
gl_mchar_t *mbs1 = (gl_mchar_t *) malloc(20*IFX_GL_MB_MAX+1);

/* dynamic allocation with ifx_gl_mb_loc_max() */
gl_mchar_t *mbs2 = (gl_mchar_t *)
malloc(20*ifx_gl_mb_loc_max()+1);
```

**Related concepts**:

"The gl_mchar_t data type" on page 2-3

## Wide-character-string allocation

When you represent a wide-character string in an array, the number of array elements does equal the number of wide characters in the string. Therefore, you can use the same allocation method for wide-character strings as for single-byte strings.

For example, the following declaration statically allocates 20 wide characters for the **wcs** string:

```
gl_wchar_t wcs[20];
```

The following declaration dynamically allocates 20 wide characters for the **wc1** string:

```
gl_wchar_t *wcs1 = (gl_wchar_t *) malloc(20*sizeof(gl_wchar_t));
```

If your wide-character string is null terminated, you must allocate one additional character for the null terminator. The null terminator requires the same space allocated as an entire wide-character. The following declaration allocates three null-terminated wide-character strings:

```
/* static allocation */
gl_wchar_t wcs[21];
/* dynamic allocation */
gl_wchar_t *wcs3 = (gl_wchar_t *) malloc(21*sizeof(gl_wchar_t));
```

**Related concepts**:

"The gl_wchar_t data type" on page 2-4

### String deallocation

The IBM Informix GLS library does not automatically deallocate memory that you dynamically allocate. Once you no longer need the string buffer, you must ensure that you deallocate any memory that your application has dynamically allocated for multibyte-character and wide-character strings.

The DataBlade API does provide some automatic garbage collection for memory that you allocate dynamically. When this memory is deallocated depends on the memory duration with which it was allocated. However, it is good programming practice to handle memory deallocation implicitly whenever possible. For more information about memory management with the DataBlade API, see the *IBM Informix DataBlade API Programmer's Guide*.

## Keep multibyte strings consistent

You must take special measures to perform *truncation* and *fragmentation* operations on multibyte strings so that you do not split a multibyte character.

**Truncation**
Makes a long character string fit into a smaller buffer.

**Fragmentation**
Breaks a long character string into two or more nonadjacent buffers to meet the memory-management requirements of their components.

### Truncate multibyte strings

Sometimes you need to truncate a long character string so that it fits into a smaller buffer. When you truncate a character string that contains just single-byte characters, you can truncate at an arbitrary byte location in the string. Because each character is one byte long, the truncated result still contains only complete characters.

However, to truncate a string that might contain even one multibyte character, you must take special measures. If you truncate at an arbitrary byte location in a multibyte-character string, you might truncate at a byte that is part of a multibyte character. In this case, the truncated string might end with only the first 1, 2, or 3 bytes of a multibyte character without the remaining bytes of the character. For such a string, subsequent traversal could result in an attempt to read beyond the end of the buffer.

Therefore, all IBM Informix GLS functions that traverse one multibyte character or a length-terminated multibyte-character string set the error number to IFX_GL_EINVAL if they detect that an otherwise valid character has been truncated.

If you know that no truncation has occurred to the string, you can consider the IFX_GL_EINVAL error the same as IFX_GL_EILSEQ. However, if truncation might have occurred, IFX_GL_EINVAL indicates that you need to further truncate the string so that the last character in the string is complete. Depending on your application, you might take one of the following actions:

- Make the truncated string even shorter than originally intended.
- Replace the first 1, 2, or 3 bytes of the truncated character with a padding character that is appropriate for your application.

**Important:** Even though the Informix GLS library functions can detect invalid characters after truncation has occurred, it is much better to avoid the situation.

## Fragment multibyte strings

Sometimes you need to fragment a long character string into two or more nonadjacent buffers to meet the memory-management requirements of their components. When you fragment a character string that contains just single-byte characters, you can fragment at an arbitrary byte location in the string. Because each character is one byte long, the fragmented results are still each a complete character string.

However, to fragment a string that might contain even one multibyte character, you must take special measures. If you fragment at an arbitrary byte location in a multibyte-character string, you might fragment at a byte that is part of a multibyte character. In this case, one fragment might end with the first 1, 2, or 3 bytes of a character, while the next fragment starts with the remaining byte or bytes.

If the only thing that you ever do with these fragments is to concatenate them back together to form one string, you do not need to perform any special processing. However, if you need to traverse the fragments as multibyte strings, these fragments might cause an attempt to read beyond the end of one fragment or an illegal character at the beginning of the next fragment.

Therefore, all IBM Informix GLS functions that traverse one multibyte character or a length-terminated multibyte-character string set the error number to IFX_GL_EINVAL if they detect an otherwise valid character at the end of a fragment.

**Important:** The Informix GLS functions cannot detect that the beginning of a fragment contains the remaining bytes of the last character in some previous fragment because they cannot look at the previous fragment first. Therefore, they might interpret the last 1, 2, or 3 bytes of a multibyte character as a valid character.

If you know that no fragmentation has occurred on the string, you can consider the IFX_GL_EINVAL error the same as IFX_GL_EILSEQ. However, if fragmentation might have occurred, IFX_GL_EINVAL indicates that you need to fragment the string so that each fragment is a complete string. Depending upon your application, you might take one of the following actions:

- Make a fragment even shorter than originally intended.
- Replace the first 1, 2, or 3 bytes of the fragmented character with a padding character that is appropriate for your application and shift these bytes to the beginning of the next fragment.

**Important:** Even though the Informix GLS library functions can detect invalid characters after fragmentation has occurred, it is much better to avoid the situation.

**Related concepts**:
"Preserve state information" on page 2-15

# Chapter 3. Data formatting

You must handle the format of locale-specific data in your applications.

These topics answer the following questions:

- What are the different locale-file categories and what locale-specific formats do they define?
- How do you use the IBM Informix GLS library to convert and format locale-specific data?

**Related reference**:

Chapter 2, "Character processing," on page 2-1

## Locale-specific data formats

The format in which data appears within an application when it is in literal strings or character variables is called the *end-user format*. End-user formats are locale specific: different countries use different formats for numeric, monetary, date, and date/time data.

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 | SQL built-in data types |
|---|---|---|
| Numeric | LC_NUMERIC | DECIMAL, INT8, INTEGER, SMALLINT, FLOAT, SMALLFLOAT |
| Monetary | LC_MONETARY | MONEY |
| Date and time | LC_TIME | DATE, DATETIME |

For more information about the end-user formats that this section describes, see the *IBM Informix GLS User's Guide*.

### The LC_NUMERIC locale-file category

The LC_NUMERIC locale-file category is the section within a GLS locale file that defines the locale-specific formats for strings that contain numeric data.

Numeric data is considered to be all values that contain digits except monetary data. Therefore, integer, fixed-point, and floating-point numbers are all considered numeric data. Strings of numeric data are called number strings.

**Tip:** In the *IBM Informix GLS User's Guide*, the LC_NUMERIC locale-file category is sometimes referred to as the NUMERIC locale category. This manual uses LC_NUMERIC as the category name for numeric data.

The LC_NUMERIC category of each locale file contains subcategories that define the different numeric formats for that locale. The following table lists the numeric formats and their corresponding LC_NUMERIC subcategory.

| Numeric format | LC_NUMERIC subcategory |
|---|---|
| Numeric decimal separator | **decimal_point** |
| Numeric thousands separator | **thousands_sep** |
| Numeric grouping information (number of numeric digits to group together before inserting a thousands separator) | **grouping** |
| Numeric positive sign | **num_positive_sign** |
| Numeric negative sign | **num_negative_sign** |

## The LC_MONETARY locale-file category

The LC_MONETARY locale-file category is the section within a GLS locale file that defines the locale-specific formats for strings that contain monetary data.

Monetary data is considered to be values that contain digits and that represent units of some currency. Therefore, only fixed-point numbers can be monetary data. All other types of numbers (such as integer and floating-point) are considered to be numeric data. Strings of monetary data are called money strings.

**Tip:** In the *IBM Informix GLS User's Guide*, the LC_MONETARY locale-file category is sometimes referred to as the MONETARY locale category. This manual uses LC_MONETARY as the category name for monetary data.

The LC_MONETARY category of each locale file contains subcategories that define the different monetary formats for that locale. The following table lists the monetary formats and their corresponding LC_MONETARY subcategory.

| Monetary format | LC_MONETARY subcategory |
|---|---|
| International currency symbol (in accordance with the ISO 4217:1987 standard) | **int_curr_symbol** |
| National (local) currency symbol | **currency_symbol** |
| Monetary decimal separator | **mon_decimal_point** |
| Monetary thousands separator | **mon_thousands_sep** |
| Monetary grouping information (number of monetary digits to group together before inserting a thousands separator) | **mon_grouping** |
| Monetary positive sign | **positive_sign** |
| Monetary negative sign | **negative_sign** |
| Number of fractional digits (those digits to the right of the decimal separator) to use in the international monetary format | **int_frac_digits** |
| Number of fractional digits (those digits to the right of the decimal separator) to use in the national (local) monetary format | **frac_digits** |
| Whether the currency symbol precedes or follows the monetary value | **p_cs_precedes**, **n_cs_precedes** |
| Whether to separate the currency symbol from the monetary value with a space | **p_sep_by_space**, **n_sep_by_space** |
| Position of positive and negative signs in a monetary value | **p_sign_posn**, **n_sign_posn** |

**Tip:** The **DBMONEY** environment variable can also specify formats for locale-specific formats for money strings.

For definitions of any of these monetary formats (such as currency symbol, decimal separator, or thousands separator) and more information about the **DBMONEY** environment variable, see the *IBM Informix GLS User's Guide*.

## The LC_TIME locale-file category

The LC_TIME locale-file category is the section within a GLS locale file that defines the locale-specific formats for strings that contain date and time data.

Such strings are called *date strings* (which contain only date data) or *date/time strings* (which contain date and time data).

**Tip:** In the *IBM Informix GLS User's Guide*, the LC_TIME locale-file category is sometimes referred to as the TIME locale category. This manual uses LC_TIME as the category name for date and time data.

The LC_TIME category of each locale file contains subcategories that define the different date and time formats. The following table lists the available formats and their corresponding LC_TIME subcategory.

| Date/time format | LC_TIME subcategory |
|---|---|
| Abbreviated weekday names | **abday** |
| Full weekday names | **day** |
| Abbreviated month names | **abmon** |
| Full month names | **mon** |
| Date representation | **d_fmt** |
| Time representation | **t_fmt** |
| Date/time representation | **d_t_fmt** |
| a.m. and p.m. equivalents | **am_pm** |
| Representation of date/time with a.m. and p.m. indicators | **t_fmt_ampm** |
| Era names | **era** |
| Representation of date with eras | **era_d_fmt** |
| Representation of time with eras | **era_t_fmt** |
| Representation of date/time with eras | **era_d_t_fmt** |
| Alternative digits to use in date and time strings | **alt_digits** |

**Tip:** The **GL_DATE**, **GL_DATETIME**, **DBDATE**, and **DBTIME** environment variables can also specify formats for locale-specific formats for date and time strings.

For definitions of any of these date or time formats (such as eras) and information about related environment variables, see the *IBM Informix GLS User's Guide*.

# Conversion and formatting with Informix GLS

The IBM Informix GLS library provides functions that allow you to perform the conversion and formatting tasks on locale-specific data.

**Conversion**

Changes a string that contains locale-specific formats 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 a locale-specific string.

You usually perform formatting of a locale-specific string to prepare the internal representation of a value for display or printing to the end user.

The internal representation of a value is one that can be stored directly in a database column. The following table lists the SQL data types for the internal representations along with the ESQL/C and DataBlade API data types that hold these internal representations.

| Type of data | SQL data type | ESQL/C data type | DataBlade API data type |
|---|---|---|---|
| Numeric | DECIMAL<br><br>* SMALLINT, INTEGER, INT8, FLOAT, SMALLFLOAT | decimal (dec_t) | mi_decimal |
| Monetary | MONEY | decimal (dec_t) | mi_decimal |
| Date | DATE | date (long int) | mi_date |
| Date and time | DATETIME | datetime (dtime_t) | mi_datetime |

* Numeric data includes integer, floating-point, and fixed-point numbers. The IBM Informix GLS conversion and formatting functions represent all these numeric data types with the decimal (or mi_decimal) data type. To convert the decimal (or mi_decimal) representation to an integer or floating-point value, use a library function such as **dectoint()**, **dectolong()**, or **dectodbl()**. For more information about these decimal-conversion functions, see the *IBM Informix ESQL/C Programmer's Manual*.

The following table lists the functions that the Informix GLS library provides for the conversion and formatting of locale-specific data.

*Table 3-1. Informix GLS conversion and formatting functions*

| Locale-specific data | Conversion | Formatting |
|---|---|---|
| Numeric | **ifx_gl_convert_number()** | **ifx_gl_format_number()** |
| Monetary | **ifx_gl_convert_money()** | **ifx_gl_format_money()** |
| Date | **ifx_gl_convert_date()** | **ifx_gl_format_date()** |
| Date/time | **ifx_gl_convert_datetime()** | **ifx_gl_format_datetime()** |

The conversion and formatting functions in the preceding table accept a *format string* to indicate how to handle a locale-specific string. A format string is composed of white spaces, ordinary characters, and one or more *formatting directives*. A formatting directive consists of the following characters:

- A percent symbol (%)
- Optional format modifiers
- A type-specifier character, which determines the type of conversion

For example, the **ifx_gl_convert_number()** function supports the following formatting directive: `%0x`

In the preceding formatting directive, 0 is a format modifier that indicates padding, and x is a type specifier that indicates the hexadecimal format of a number.

## Convert a locale-specific string

The IBM Informix GLS conversion functions scan the incoming locale-specific string to create a corresponding internal representation, as follows.

| | | Converted form | |
|---|---|---|---|
| **Conversion function** | **Unconverted form** | **ESQL/C data type** | **DataBlade API data type** |
| **ifx_gl_convert_number()** | Number string | decimal (dec_t) | mi_decimal |
| **ifx_gl_convert_money()** | Money string | decimal (dec_t) | mi_decimal |
| **ifx_gl_convert_date()** | Date string | date (long int) | mi_date |
| **ifx_gl_convert_datetime()** | Date/time string | datetime (dtime_t) | mi_datetime |

The formatting directives in the format string tell the conversion functions what conversions to perform on the locale-specific string. To process a formatting directive, a conversion function consults the appropriate category in the current locale to obtain any locale-specific formats and then converts the resulting value to an internal representation that can be stored in a database.

For example, if the current locale is the default locale, the following DataBlade API call to the **ifx_gl_convert_number()** function converts the number string `"1,450"` to its hexadecimal equivalent in an **mi_decimal** value:

```
mi_decimal num_val;
...
if ( ifx_gl_convert_number(&num_val, "1,450","%0x") != 0 )
    /* handle error */
```

In the current locale, the thousands separator is defined as the comma (,) symbol. Therefore, the **ifx_gl_convert_number()** function must correctly interpret the comma in the number string so that it can convert this string to the hexadecimal equivalent of the value 1450 and store the result in the **mi_decimal** value, **num_val**.

If the current locale is French (**fr_fr**), the thousands separator is a space. Therefore, the following call to **ifx_gl_convert_number()** must interpret a space as the thousands separator to convert the French number string (`"1 450"`) successfully to its **mi_decimal** equivalent:

```
mi_decimal *num_val;
...
if ( ifx_gl_convert_number(num_val, "1 450","%0x") != 0 )
    /* handle error */
```

## Format a locale-specific string

The IBM Informix GLS conversion functions scan the incoming locale-specific string to create a corresponding internal representation, as follows.

| Formatting function | Unformatted form | | Formatted form |
|---|---|---|---|
| | ESQL/C internal representation | DataBlade API internal representation | |
| **ifx_gl_format_number()** | decimal (dec_t) | mi_decimal | Number string |
| **ifx_gl_format_money()** | decimal (dec_t) | mi_decimal | Money string |
| **ifx_gl_format_date()** | date (long int) | mi_date | Date string |
| **ifx_gl_format_datetime()** | datetime (dtime_t) | mi_datetime | Date/time string |

The formatting directives in the format string tell the formatting functions how to format the internal representation of a value into a locale-specific string. To process a formatting directive, a formatting function consults the appropriate category in the current locale to obtain any locale-specific formats and then formats the locale-specific string with this information.

For example, the following DataBlade API call to the function **ifx_gl_format_number()** converts the hexadecimal representation of the number 1450 to a number string:

```
mi_decimal num_val;
char num_str[BUFSIZE];
...
if ( ifx_gl_format_number(num_str, BUFSIZE, &num_val,"%0x") != 0 )
    /* handle error */
```

If the current locale is the default locale, **num_str** contains the number string `"1,450"` upon successful completion of this call to **ifx_gl_format_number()**. The function correctly formats the comma (,) as the thousands separator in the number string. If the current locale is French (**fr_fr**), the thousands separator is a space. Therefore, the preceding call to the function **ifx_gl_convert_number()** would format the French number string as `"1 450"`.

# Chapter 4. Informix GLS functions

These topics describe the functions that the IBM Informix GLS library provides. First is a summary of functions by categories. The rest of the topics provide reference information about the Informix GLS public functions, in alphabetical order.

For information about DataBlade API header files and data structures to use with the functions, see the *IBM Informix DataBlade API Programmer's Guide*.

**Related concepts**:

"Character operations" on page 2-5

**Related reference**:

"Using Informix GLS in a C-language program" on page 1-4

"String operations" on page 2-18

## Function summary

This topic groups all the functions of the IBM Informix GLS library by task.

### Memory allocation

The Informix GLS library provides the following macro and function to assist in memory allocation:

- The IFX_GL_MB_MAX macro
- The **ifx_gl_mb_loc_max()** function

For more information, see "Allocate memory" on page 1-10.

### Initialization and error handling

The Informix GLS library provides the following functions:

- The **ifx_gl_init()** function initializes the Informix GLS library.

  For more information, see "Initialize the Informix GLS library" on page 1-8.
- The **ifx_gl_lc_errno()** function obtains the value of the Informix GLS error number.

  For more information, see "Informix GLS exceptions" on page 1-8.

### Stream input and output

The Informix GLS library provides the **ifx_gl_getmb()** and **ifx_gl_putmb()** functions to input and output multibyte characters in a user-defined location.

For more information, see "Input and output streams" on page 1-10.

### Character processing

Most of the Informix GLS functions provide support for character processing of both multibyte and wide characters. This section lists the character-processing tasks and the Informix GLS functions that support them. For general information about character processing, see Chapter 2, "Character processing," on page 2-1.

### Code-set conversion

The Informix GLS library provides the following functions to perform code-set conversion:

- **ifx_gl_conv_needed()**
- **ifx_gl_cv_mconv()**
- **ifx_gl_cv_outbuflen()**
- **ifx_gl_cv_sb2sb_table()**

For general information, see "Code-set conversion" on page 2-13.

### Character classification

The Informix GLS library provides the following functions for character classification of multibyte and wide characters.

| Multibyte-character classification | Wide-character classification |
|---|---|
| **ifx_gl_ismalnum()** | **ifx_gl_iswalnum()** |
| **ifx_gl_ismalpha()** | **ifx_gl_iswalpha()** |
| **ifx_gl_ismblank()** | **ifx_gl_iswblank()** |
| **ifx_gl_ismcntrl()** | **ifx_gl_iswcntrl()** |
| **ifx_gl_ismdigit()** | **ifx_gl_iswdigit()** |
| **ifx_gl_ismgraph()** | **ifx_gl_iswgraph()** |
| **ifx_gl_ismlower()** | **ifx_gl_iswlower()** |
| **ifx_gl_ismprint()** | **ifx_gl_iswprint()** |
| **ifx_gl_ismpunct()** | **ifx_gl_iswpunct()** |
| **ifx_gl_ismspace()** | **ifx_gl_iswspace()** |
| **ifx_gl_ismupper()** | **ifx_gl_iswupper()** |
| **ifx_gl_ismxdigit()** | **ifx_gl_iswxdigit()** |

For general information, see "Character classification" on page 2-5.

### Case conversion

The Informix GLS library provides the following functions for case conversion of multibyte and wide characters.

| Multibyte-character case conversion | Wide-character case conversion |
|---|---|
| **ifx_gl_tomlower()** | **ifx_gl_towlower()** |
| **ifx_gl_tomupper()** | **ifx_gl_toupper()** |
| **ifx_gl_mbsspn()** | **ifx_gl_wcsspn()** |

In addition, the Informix GLS library provides the function **ifx_gl_case_conv_outbuflen()**. For general information, see "Case conversion" on page 2-8.

### String traversal

The Informix GLS library provides the following functions for string traversal of multibyte-character strings:

- **ifx_gl_mblen()**

- **ifx_gl_mbsnext()**
- **ifx_gl_mbsprev()**

For general information, see "String traversal" on page 2-18.

**String processing**

The Informix GLS library provides the following functions for string processing of multibyte and wide characters.

| Multibyte-character classification | Wide-character classification |
|---|---|
| **ifx_gl_mbscat()** | **ifx_gl_wcscat()** |
| **ifx_gl_mbschr()** | **ifx_gl_wcschr()** |
| **ifx_gl_mbscpy()** | **ifx_gl_wcscpy()** |
| **ifx_gl_mbscspn()** | **ifx_gl_wcscspn()** |
| **ifx_gl_mbslen()** | **ifx_gl_wcslen()** |
| **ifx_gl_mbsmbs()** | **ifx_gl_wcswcs()** |
| **ifx_gl_mbsncat()** | **ifx_gl_wcsncat()** |
| **ifx_gl_mbsncpy()** | **ifx_gl_wcsncpy()** |
| **ifx_gl_mbsntslen()** | **ifx_gl_wcsntslen()** |
| **ifx_gl_mbsntsbytes()** | None |
| **ifx_gl_mbspbrk()** | **ifx_gl_wcspbrk()** |
| **ifx_gl_mbsrchr()** | **ifx_gl_wcsrchr()** |
| **ifx_gl_mbsspn()** | **ifx_gl_wcsspn()** |

For general information, see "String processing" on page 2-19.

**Character/string comparison and sorting**

The Informix GLS library provides the **ifx_gl_mbscoll()** and **ifx_gl_wcscoll()** functions to obtain locale-specific order for characters and strings.

For general information, see "Character/string comparison and sorting" on page 2-20.

## Data formatting

The Informix GLS library provides functions to support the conversion and formatting of locale-specific strings. This section lists the data types that involve locale-specific strings and the Informix GLS functions that support them. For general information about data formatting, see Chapter 3, "Data formatting," on page 3-1.

**Date and time conversion and formatting functions**

The Informix GLS library provides the following functions for conversion and formatting of date and date/time strings:

- **ifx_gl_convert_date()**
- **ifx_gl_convert_datetime()**
- **ifx_gl_format_date()**
- **ifx_gl_format_datetime()**

**Numeric conversion and formatting**

The Informix GLS library provides the **ifx_gl_convert_number()** and **ifx_gl_format number()** functions for conversion and formatting of number strings.

**Money conversion and formatting**

The Informix GLS library provides the **ifx_gl_convert_money()** and **ifx_gl_format_money()** functions for conversion and formatting of money strings.

# Function reference

This section describes the syntax, usage, and return values of IBM Informix GLS functions in alphabetical order.

## The ifx_gl_case_conv_outbuflen() function

The **ifx_gl_case_conv_outbuflen()** function calculates an approximation of the number of bytes required to store a case-converted multibyte character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_case_conv_outbuflen(src_mbs_bytes)
       int src_mbs_bytes;
```

*src_mbs_bytes*
The integer number of bytes in the buffer of multibyte characters to be case converted.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_case_conv_outbuflen()** function returns one of the following values:

- The exact number of bytes that a buffer of case-equivalent multibyte characters requires
- A close over-approximation of this number of bytes

This buffer can contain one or more multibyte characters. This function applies to both uppercase and lowercase conversions.

Use this function to determine whether case conversion of multibyte characters can be performed in place. If the value that this function returns is not equal to *src_mbs_bytes*, case conversion of multibyte characters cannot be performed in place. You must allocate a separate multibyte destination buffer. However, if the value that this function returns is exactly equal to *src_mbs_bytes*, multibyte case conversion can be performed in place.

**Tip:** This function does not apply to wide-character case conversions.

### Return values

**>0** The number of bytes required to store multibyte characters of length *src_mbs_bytes* after they have been converted to either lowercase or uppercase characters.

**0** The function was not successful.

### Errors

None

**Related reference**:

"The ifx_gl_tomlower() function" on page 4-114

"The ifx_gl_tomupper() function" on page 4-116

## The ifx_gl_complen() function

The **ifx_gl_complen()** function determines the length of a collating element.

### Syntax

```
#include <gls.h>
int ifx_gl_complen(gl_lc_t lc, gl_mchar_t *input)
```

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_complen()** function is used for string scanning. Skipping to the next character in an input string is achieved by calling this function with the current character as an argument and adding to it the return value of the function.

The **ifx_gl_complen()** function returns the length in bytes of the initial part of an input string that matches a collating element. Multicharacter collation sequences are special sequences of two or more characters that in a specific local take a different meaning. For example in any es_es locale, if the initial part of an input string begins with `c` or `ca`, the function returns 0 to indicate that the input string is not a multicharacter collating element. If the initial part of the input string begins with `ch`, the function returns 2 to indicate that the input string is a multicharacter collating element.

If the **ifx_gl_complen()** function returns 0, other GLS functions, such as **ifx_gl_mbslen()**, are required to determine the length of the current character in the input string.

### Return values

**0** The initial part of the input string is not a multicharacter collating element.

**2 and above**
The initial part of the input string is a multicharacter collating element

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**GL_INVALIDLOC**
An invalid locale is passed to the function.

**GL_ENULLPTR**
The input is null.

**GL_EILSEQ**
The input is an invalid character in the locale.

**GL_ENOSYS**
The operation on locale is invalid.

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbslen() function" on page 4-95

## The ifx_gl_conv_needed() function

The **ifx_gl_conv_needed()** function determines whether code-set conversion between two code sets is required.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_conv_needed(dst_codeset, src_codeset)
       char *dst_codeset;
       char *src_codeset;
```

*dst_codeset*
A pointer to the name of the destination (target) code set.

*src_codeset*
A pointer to the name of the source code set.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_conv_needed()** function determines whether characters encoded in *src_codeset* require conversion to *dst_codeset*. It is not enough to compare the names of the code sets because several names can identify the same code set. For example, 8859-1, 819, and Latin-1 refer to the same code set.

The code sets, *src_codeset* and *dst_codeset*, can be any of the following:

- Locale names
- Code-set names
- The IFX_GL_PROC_CS macro

### Return values

**0** Code-set conversion is not needed between the code sets that *dst_codeset* and *src_codeset* specify.

**1** Code-set conversion is needed between the code sets that *dst_codeset* and *src_codeset* specify.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, the function sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EINVAL**
The *src_codeset* string is not a valid locale specification or code-set name, or the code-set name could not be mapped to a code-set number.

**IFX_GL_EBADF**
Function cannot find the code-set registry.

**IFX_GL_BADFILEFORM**
Bad format found in the code-set registry.

**Related reference**:

"The ifx_gl_cv_mconv() function" on page 4-26

"The ifx_gl_cv_outbuflen() function" on page 4-29

"The ifx_gl_cv_sb2sb_table() function" on page 4-30

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_convert_date() function

The **ifx_gl_convert_date()** function converts a date string to its internal date representation.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_convert_date(date, datestr, format)
       mi_date *date;
       char *datestr;
       char *format;
```

*date* A pointer to the variable that holds the internal date representation that **ifx_gl_convert_date()** creates from the *datestr* date string.

*datestr* A pointer to the first character of the date string that the function converts to its internal date representation.

*format* A pointer to the format string that determines how to interpret the *datestr* date string. For more information, see "Format string" on page 4-8.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_convert_date()** function converts the date string that *datestr* references to its internal date representation, which the function stores in the *date* argument. The function uses the format that the *format* string specifies to scan the *datestr* date string.

The *date* argument is a pointer to a **date (4-byte int)** value.

The *date* argument is a pointer to an **mi_date** value.

## Format string

If *format* is NULL, the **ifx_gl_convert_date()** function determines the format of the *datestr* date string that it scans from the environment, as follows:

1. If the **DBDATE** environment variable is set, the function scans *datestr* according to the order of the format elements in **DBDATE**.
2. If the **GL_DATE** environment variable is set, the function scans *datestr* according to the specification of **GL_DATE**.
3. Otherwise, the function obtains the format from the **d_fmt** subcategory of the LC_TIME category in the current GLS locale file.

If *format* is not NULL, it must point to a string that follows the rules that this section describes. To convert the *datestr* date string, **ifx_gl_convert_date()** compares each character in *datestr* with the *format* string. It takes the following actions for each possible character that it finds in *format*.

| Contents of format string | Conversion action taken |
|---|---|
| One or more white space characters | The function skips over the corresponding number of white space characters in the *datestr* date string (unless the formatting directive begins with the minus-sign modifier), up to the first character that is not white space or until no more characters can be scanned. White space characters are characters that the blank class of the LC_CTYPE category in the current locale defines.<br><br>To execute a series of formatting directives composed of **%n**, **%t**, white space characters, or any combination, the function scans up to the first character that is not white space (which remains unscanned) or until no more characters can be scanned. |
| A valid formatting directive | The function performs the specified conversion on the date element in the *datestr* date string. It replaces the formatting directive with an internal representation of the date element for conversion to the *date* value. There must be white space or other nonalphanumeric characters between any two formatting directives. |
| Ordinary characters | The function must find the same ordinary character in the *datestr* date string. Any mismatch generates an error. The differing and subsequent characters in *datestr* remain unscanned.<br><br>You cannot include white space characters as ordinary characters. |

The formatting directive consists of the following sequence:

*%[modifiers][flags][maximum_width][.minimum_width]type_specifier*

| Argument | See |
|---|---|
| *modifiers* | "Modified formatting directives" on page 4-10 |
| *flags* | "Field width" on page 4-11 |
| *minimum_width* | "Field width" on page 4-11 |
| *maximum_width* | "Field width" on page 4-11 |
| *type_specifier* | "Valid type specifiers" |

**Tip:** In the preceding format sequence, the square brackets indicate that the enclosed portion of the format is optional.

## Valid type specifiers

The type specifier is a letter (or letters) within a formatting directive that identifies a format for the **ifx_gl_convert_date()** function to expect when it scans a date element of a *datestr* string. These date formats are formats that the LC_TIME locale-file category of the current locale might define. The **ifx_gl_convert_date()** function supports the following formatting directives to represent a date element.

**%a** Matches the day of the week. You can specify the abbreviated weekday name, which the **abday** subcategory of the LC_TIME defines or specify the full weekday name, which the **day** subcategory of LC_TIME defines.

**%A** Same as **%a**.

**%b** Matches the month. You can specify the abbreviated month name, which the **abmon** subcategory of LC_TIME defines or specify the full month name, which the **mon** subcategory of LC_TIME defines.

**%B** Same as **%b**.

**%C** Matches the century number in the range 0 - 99. Leading zeros are permitted but not required. If **%C** is used without **%y**, it is ignored.

**%d** Matches the day of the month as a decimal number in the range 1 - 31. Leading zeros are permitted but not required.

**%e** Same as **%d**.

**%h** Same as **%b**.

**%j** Matches the day of the year as a decimal number in the range 1 - 366. Leading zeros are permitted but not required.

**%m** Matches the month as a decimal number in the range 1 - 12. Leading zeros are permitted but not required.

**%n** Matches any horizontal white space that the blank class of the LC_CTYPE category defines.

**%t** Matches any vertical white space that the space class of the LC_CTYPE category defines.

**%u** Matches the weekday as a decimal number in the range 1 -7, with 0 representing Sunday. Leading zeros are permitted but not required.

**%w** Matches the weekday as a decimal number in the range 0 - 6, with 0 representing Sunday. Leading zeros are permitted but not required.

**%x** Indicates use of the format that the **d_fmt** subcategory of LC_TIME defines.

**%y** Matches the year within century as a decimal number in the range 0 - 99. Leading zeros are permitted but not required. If **%y** is used without **%C** and the month and day of month are part of the *datestr* string, the function determines the century from the **DBCENTURY** environment variable.

**%Y** Matches the year, including the century, as a decimal number in the range 0 - 9999.

**%%** Matches the % symbol.

**Tip:** The **ifx_gl_convert_date()** function ignores case when it matches items such as month or weekday names.

If the *format* string contains redundant formatting directives, directives that are closer to the end of the *format* string take precedence over the directives that are closer to the beginning of the *format* string.

If a formatting directive does not correspond to any of the preceding directives, the behavior of the conversion is undefined.

## Modified formatting directives

You can modify some formatting directives with format modifiers, which follow the percent symbol (%), to indicate use of an alternative format that the LC_TIME locale-file category of the current locale might define. The **ifx_gl_convert_date()** function supports the following format modifiers.

**E** Use formats that include the era-based dates.

LC_TIME Subcategory: **era**, **era_d_fmt**

**O** Use alternative locale-specific digits in dates.

LC_TIME Subcategory: **alt_digits**

The alternative format replaces one that an unmodified formatting directive normally uses. If the alternative format does not exist for the current locale, the behavior is the same as if unmodified formatting directives were used. The **ifx_gl_convert_date()** function supports the following modified formatting directives with the **E** and **O** modifier.

**%EC** Matches the name of the base year (period) in the alternative representation. You can specify either the abbreviated or full name, which the **era** subcategory of LC_TIME defines.

**%Eg** Same as **%EC**.

**%Ex** Indicates use of the format that the **era_d_fmt** subcategory of LC_TIME category defines.

**%Ey** Matches the offset from **%EC** (year only) in the alternative representation, which the **era** subcategory of LC_TIME defines.

**%EY** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the full alternative year representation, which the **era** subcategory of LC_TIME defines.

**%Od** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the day of the month. Leading zeros are permitted but not required.

**%Oe** Same as **%Od**.

**%Om** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the month.

**%Ow** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the weekday as a number (Sunday=0).

**%Oy** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the value of **%Ey**.

The **ifx_gl_convert_date()** function needs the era base, era offset, day, and month to determine the year from era information. The function uses a default era offset of 1 if you specify era base, day, and month but not era offset.

The **ifx_gl_convert_date()** function also supports the **i** modifier in the following formatting directives to support formats that are compatible with earlier IBM Informix date and time formatting.

**%iy** Indicates use of the Informix **DBDATE** format Y2. Both 98 and 1998 are interpreted as 1998 (or the century that **DBCENTURY** indicates).

**%iY** Indicates use of the Informix **DBDATE** format Y4. Both 98 and 1998 are interpreted as 1998 (or the century that **DBCENTURY** indicates).

## Field width

You can modify some formatting directives with the following modifiers to indicate use of an alternative format that the LC_TIME locale-file category of the current locale might define:

[- | 0][*maximum_width*][.*minimum_width*]

**[- | 0]**
Indicates field justification.

If the specification begins with a minus sign (-), the function assumes that the field is left-aligned. In this case, the value that is being read must start with a digit and can be trailed with spaces. The field can be preceded with leading spaces unless the first character is 0. If the specification begins with 0, the function assumes that the field is right-aligned. Any padding to the left must use zeros. Otherwise, the function assumes that the *datestr* string is right-aligned.

*maximum_width*
A decimal number that indicates an optional maximum field width.

*minimum_width*
Indicates an optional minimum field width. The minimum field width has the format of a period (.) followed by a decimal number. The *minimum_width* decimal number represents the minimum number of characters to read. If the function reads fewer than *minimum_width* characters, it generates an error (unless you specified left justification). When you specify left justification, the function skips any trailing white space to read the required number of characters.

## Locale information

The LC_TIME category of the current locale affects the behavior of this function because it provides the locale-specific information for the scan of the *datestr* date string. For more information, see "The LC_TIME locale-file category" on page 3-3.

## Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurs, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_E2BIG**
Operation would overflow buffer.

**IFX_GL_EBADF**
A formatting directive is invalid.

**IFX_GL_INVALIDFMT**
The format string is invalid.

**IFX_GL_EDAYRANGE**
Day number is out of bounds.

**IFX_GL_EWKDAYRANGE**
Weekday number is out of bounds.

**IFX_GL_EYDAYRANGE**
Yearday number is out of bounds.

**IFX_GL_EMONTHRANGE**
Month number is out of bounds.

**IFX_GL_EYEARRANGE**
Year number is out of bounds.

**IFX_GL_EERAOFFRANGE**
Era offset is out of bounds.

**IFX_GL_BADDAY**
Month (as a number) could not be scanned.

**IFX_GL_BADWKDAY**
Weekday (as a number) could not be scanned.

**IFX_GL_BADYDAY**
Day of year (as a number) could not be scanned.

**IFX_GL_BADMONTH**
Month could not be scanned.

**IFX_GL_BADYEAR**
Year could not be scanned.

**IFX_GL_BADERANAME**
Era name is invalid.

**IFX_GL_BADERAOFFSET**
Era offset is invalid.

**IFX_GL_BADFMTMOD**
Format modifier is invalid.

**IFX_GL_BADFMTWP**
Width is invalid.

**IFX_GL_BADINPUT**
Input string does not match format string.

**IFX_GL_NOPOINT**
Missing decimal point in input string.

**IFX_GL_BADMONTHSTR**
Month string could not be scanned.

**IFX_GL_BADERASPEC**
Could not load era from locale.

**Related reference**:

"The ifx_gl_convert_datetime() function"

"The ifx_gl_format_date() function" on page 4-31

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_convert_datetime() function

The **ifx_gl_convert_datetime()** function converts a date/time string to its internal date/time representation.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_convert_datetime(datetime, datetimestr, format)
       mi_datetime *datetime;
       char *datetimestr;
       char *format;
```

*datetime*
A pointer to the variable that holds the internal date/time representation that **ifx_gl_convert_datetime()** creates from the *datetimestr* date/time string.

*datetimestr*
A pointer to the first character of the date/time string that the function converts to its internal date/time representation.

*format* A pointer to the format string that determines how to interpret the *datetimestr* date/time string. For more information, see "Format string" on page 4-8.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_convert_datetime()** function converts the date/time string that *datetimestr* references to its internal date/time representation, which the function stores in the *datetime* argument. The function uses the format that the *format* string specifies to scan the *datetimestr* date/time string.

The *datetime* argument is a pointer to a **datetime (dtime_t)** value.

The *datetime* argument is a pointer to an **mi_datetime** value.

## Format string

If *format* is NULL, the **ifx_gl_convert_datetime()** function determines the format of the *datetimestr* date/time string that it scans from the environment, as follows:

1. If the **DBTIME** environment variable is set, the function scans according to **DBTIME**.
2. If the **GL_DATETIME** environment variable is set, the function scans *datetimestr* according to the specification of **GL_DATETIME**.
3. Otherwise, the function obtains the format from the **d_t_fmt** subcategory of the LC_TIME category in the current GLS locale file.

If *format* is not NULL, it must point to a string that follows the rules that this section describes. To convert the *datetimestr* date/time string, **ifx_gl_convert_datetime()** compares each character in *datetimestr* with the *format* string. It takes the following actions for each possible character that it finds in *format*.

| Contents of format string | Conversion action taken |
|---|---|
| One or more white space characters | The function skips over the corresponding number of white space characters in the *datetimestr* date/time string (unless the formatting directive begins with the minus-sign modifier), up to the first character that is not white space or until no more characters can be scanned. White space characters are characters that the blank class of the LC_CTYPE category in the current locale defines.<br><br>To execute a series of formatting directives composed of **%n**, **%t**, white space characters, or any combination, the function scans up to the first character that is not white space (which remains unscanned) or until no more characters can be scanned. |
| A valid formatting directive | The function performs the specified conversion on the date/time element in the *datetimestr* date/time string. It replaces the formatting directive with an internal representation of the date/time element for conversion to the *datetime* value. There must be white space or other nonalphanumeric characters between any two formatting directives. |
| Ordinary characters | The function must find the same ordinary character in the *datetimestr* date/time string. Any mismatch generates an error. The differing and subsequent characters in *datetimestr* remain unscanned.<br><br>You cannot include white space characters as ordinary characters. |

The formatting directive consists of the following sequence:

*%[modifiers][flags][maximum_width][.minimum_width]type_specifier*

| Argument | See |
|---|---|
| *modifiers* | "Modified formatting directives" on page 4-16 |
| *flags* | "Field width" on page 4-18 |
| *minimum_width* | "Field width" on page 4-18 |
| *maximum_width* | "Field width" on page 4-18 |
| *type_specifier* | "Valid type specifiers" |

**Tip:** In the preceding format sequence, the square brackets indicate that the enclosed portion of the format is optional.

## Valid type specifiers

The type specifier is a letter (or letters) within a formatting directive that identifies a format for the **ifx_gl_convert_datetime()** function to expect when it scans a date/time element of a *datetimestr* string. These date/time formats are formats that the LC_TIME locale-file category of the current locale might define. The **ifx_gl_convert_datetime()** function supports the following formatting directives to represent a date/time element.

**%a** Matches the day of the week. You can specify the abbreviated weekday name, which the **abday** subcategory of the LC_TIME defines or specify the full weekday name, which the **day** subcategory of LC_TIME defines. The directive ignores case when it matches weekday names.

**%A** Same as **%a**.

**%b** Matches the month. You can specify the abbreviated month name, which the **abmon** subcategory of LC_TIME defines or specify the full month name, which the **mon** subcategory of LC_TIME defines. The directive ignores case when it matches month names.

**%B** Same as **%b**.

**%c** Indicates use of the format that the **d_t_fmt** subcategory of the LC_TIME defines for the scan.

**%C** Matches the century number in the range 0 - 99. Leading zeros are permitted but not required. If **%C** is used without **%y**, it is ignored.

**%d** Matches the day of the month as a decimal number in the range 1 - 31. Leading zeros are permitted but not required.

**%D** Is the same as **%m/%d/%y**.

**%e** Same as **%d**.

**%F[***n***]** Matches the microsecond as a decimal number in the range 0 - 999999. Leading zeros are permitted but not required. An optional precision specification can follow the **%F**. This *n* value must be 1 - 5.

**%h** Same as **%b**.

**%H** Matches the hour (24-hour clock) as a decimal number in the range 0 - 23. Leading zeros are permitted but not required.

**%I** Matches the hour (12-hour clock) as a decimal number in the range 0 - 12. Leading zeros are permitted but not required. If **%I** is used without **%p**, the function assumes a.m.

**%j** Matches the day of the year as a decimal number in the range 1 - 366. Leading zeros are permitted but not required.

**%M** Matches the minute as a decimal number in the range 0 - 59. Leading zeros are permitted but not required.

**%n** Matches any white space that the blank class of the LC_CTYPE category defines.

**%p** Matches the equivalent of either a.m. or p.m. that the **am_pm** subcategory of LC_TIME defines.

**%r** Indicates use of the format that the **t_fmt_ampm** subcategory of LC_TIME defines.

**%R** Matches the time as **%H:%M**.

**%S** Matches the second as a decimal number in the range 0- 61. Leading zeros are permitted but not required.

**%t** Matches any white space that the space class of the LC_CTYPE category defines.

**%T** Matches the time as **%H:%M:%S**.

**%u** Matches the weekday as a decimal number in the range 1- 7, with 0 representing Sunday. Leading zeros are permitted but not required.

**%w** Matches the weekday as a decimal number in the range 0 - 6, with 0 representing Sunday. Leading zeros are permitted but not required.

**%x** Indicates use of the format that the **d_fmt** subcategory of LC_TIME defines.

**%X** Indicates use of the format that the **t_fmt** subcategory of LC_TIME defines.

**%y** Matches the year within century as a decimal number in the range 0 - 99. Leading zeros are permitted but not required. If **%y** is used without **%C** and the month and day of month are part of the *datetimestr* string, the function determines the century from the **DBCENTURY** environment variable.

**%Y** Matches the year, including the century, as a decimal number in the range 0 - 9999.

**%%** Matches the % symbol.

**Tip:** The **ifx_gl_convert_datetime()** function ignores case when it matches items such as month or weekday names.

If the *format* string contains redundant formatting directives, directives that are closer to the end of the *format* string take precedence over the directives that are closer to the beginning of the *format* string.

If a formatting directive does not correspond to any of the preceding directives, the behavior of the conversion is undefined.

## Modified formatting directives

You can modify some formatting directives with format modifiers, which follow the percent symbol (%), to indicate use of an alternative format that the LC_TIME

locale-file category of the current locale might define. The **ifx_gl_convert_datetime()** function supports the following format modifiers.

**E** Use formats that include the era-based dates and times.

LC_TIME Subcategory: **era**, **era_d_fmt**, **era_t_fmt**, **era_d_t_fmt**

**O** Use alternative locale-specific digits in dates and times.

LC_TIME Subcategory: **alt_digits**

The alternative format replaces one that an unmodified formatting directive normally uses. If the alternative format does not exist for the current locale, the behavior is the same as if unmodified formatting directives were used. The **ifx_gl_convert_datetime()** function supports the following modified formatting directives with the **E** and **O** modifier.

**%Ec** Indicates use of the format that the **era_d_t_fmt** subcategory of LC_TIME defines.

**%EC** Matches the name of the base year (period) in the alternative representation. You can specify either the abbreviated or full name, which the **era** subcategory of LC_TIME defines.

**%Eg** Same as **%EC**.

**%Ex** Indicates use of the format that the **era_d_fmt** subcategory of LC_TIME category defines.

**%EX** Indicates use of the format that the **era_t_fmt** subcategory of LC_TIME defines.

**%Ey** Matches the offset from **%EC** in the alternative representation, which the **era** subcategory of LC_TIME defines.

**%EY** Matches the full alternative year representation that the **era** subcategory of LC_TIME defines.

**%Od** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the day of the month. Leading zeros are permitted but not required.

**%Oe** Same as **%Od**.

**%OH** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the hour (24-hour clock).

**%OI** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the hour (12-hour clock).

**%Om** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the month.

**%OM** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the minutes.

**%OS** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the seconds.

**%Ow** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the weekday as a number (Sunday=0).

**%Oy** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to match the value of **%Ey**.

The **ifx_gl_convert_datetime()** function needs the era base, era offset, day, and month to determine the year from era information. The function uses a default era offset of 1 if you specify era base, day, and month but not era offset.

The **ifx_gl_convert_datetime()** function also supports the **i** modifier in the following formatting directives to support formats that are compatible with earlier IBM Informix date and time formatting.

**%iF[***n***]** Indicates use of the Informix **DBTIME** format **%F**. An optional precision specification, *n*, can follow **F**. This n value must be 1 - 5.

**%iy** Indicates use of the Informix **DBDATE** format Y2. Both 98 and 1998 are interpreted as 1998 (or the century that **DBCENTURY** indicates).

**%iY** Indicates use of the Informix **DBDATE** format Y4. Both 98 and 1998 are interpreted as 1998 (or the century that **DBCENTURY** indicates).

## Field width

You can specify an optional field width in a formatting directive. This information follows the initial percent symbol (%) of the formatting directive and has the following format:

```
[- | 0][maximum_width][.minimum_width]
```

**[- | 0]**
Indicates field justification.

If the specification begins with a minus sign (-), the function assumes that the field is left-aligned. In this case, the value that is being read must start with a digit and can be trailed with spaces. The field can be preceded with leading spaces unless the first character is 0. If the specification begins with 0, the function assumes that the field is right-aligned. Any padding to the left must use zeros. Otherwise, the function assumes that the *datetimestr* string is right-aligned.

*maximum_width*
A decimal value that specifies the maximum number of characters to read from the *datetimestr* date/time string.

*minimum_width*
Indicates an optional minimum field width. The minimum field width has the format of a period (.) followed by a decimal number. The *minimum_width* decimal number represents the minimum number of characters to read. If the function reads fewer than *minimum_width* characters, it generates an error (unless you specified left justification). When you specify left justification, the function skips any trailing white space to read the required number of characters.

**Tip:** The **ifx_gl_convert_datetime()** function ignores any field-width specification for any nonnumeric or compound formats.

For the **%F*n*** format, *n* overrides the minimum field width. If *n* is greater than the maximum field width, the maximum field width is increased to *n*. If *n* is not 1 - 5, an error is returned.

## Locale information

The LC_TIME category of the current locale affects the behavior of this function because it provides the locale-specific information for the scan of the *datetimestr* date/time string.

## Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurs, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EBADF**
A formatting directive is invalid.

**IFX_GL_EFRACRANGE**
Fraction of second is out of bounds.

**IFX_GL_ESECONDRANGE**
Second is out of bounds.

**IFX_GL_EHOURRANGE**
Hour is out of bounds.

**IFX_GL_EMINUTERANGE**
Minute is out of bounds.

**IFX_GL_EDAYRANGE**
Day number is out of bounds.

**IFX_GL_EWKDAYRANGE**
Weekday number is out of bounds.

**IFX_GL_EYDAYRANGE**
Year day number is out of bounds.

**IFX_GL_EMONTHRANGE**
Month number is out of bounds.

**IFX_GL_EYEARRANGE**
Year number is out of bounds.

**IFX_GL_EERAOFFRANGE**
Era offset is out of bounds.

**IFX_GL_BADFRAC**
Fraction could not be scanned.

**IFX_GL_BADSECOND**
Second could not be scanned.

**IFX_GL_BADMINUTE**
Minute could not be scanned.

**IFX_GL_BADHOUR**
Hour could not be scanned.

**IFX_GL_BADDAY**
Month (as a number) could not be scanned.

**IFX_GL_BADWKDAY**
Weekday (as a number) could not be scanned.

**IFX_GL_BADYDAY**
Day of year (as a number) could not be scanned.

**IFX_GL_BADMONTH**
Month could not be scanned.

**IFX_GL_BADYEAR**
Year could not be scanned.

**IFX_GL_BADERANAME**
Era name was not found.

**IFX_GL_BADERAOFFSET**
Era offset could not be scanned.

**Related concepts**:

"The LC_TIME locale-file category" on page 3-3

**Related reference**:

"The ifx_gl_convert_date() function" on page 4-7

"The ifx_gl_format_datetime() function" on page 4-37

"The ifx_gl_lc_errno() function" on page 4-85

# The ifx_gl_convert_money() function

The **ifx_gl_convert_money()** function converts a money string to its internal money representation.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_convert_money(money, monstr, format)
       mi_money *money;
       char *monstr;
       char *format;
```

*money* A pointer to the variable that holds the internal money representation that **ifx_gl_convert_money()** creates from the *monstr* money string.

*monstr* A pointer to the first character of the money string that the function converts to its internal money representation.

*format* A pointer to the format string that determines how to interpret the *monstr* money string. For more information, see "Format string" on page 4-21

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_convert_money()** function converts the money string that *monstr* references to its internal money representation, which the function stores in the *money* argument. The function uses the format that the *format* string specifies to scan the *monstr* money string.

The *money* argument is a pointer to a **decimal (dec_t)** value.

The *money* argument is a pointer to an **mi_money** value.

## Format string

If *format* is NULL, the **ifx_gl_convert_money()** function determines the format of the *monstr* money string that it scans from the environment, as follows:

1. If the **DBMONEY** environment variable is set, the function scans *monstr*, as **DBMONEY** specifies.
2. Otherwise, the function uses the formats in the LC_MONETARY category of the current GLS locale to scan the *money* value.

If *format* is not NULL, it must point to a string that follows the rules that this section describes. To convert the *monstr* money string, the function **ifx_gl_convert_money()** compares each character in *monstr* with the *format* string. It takes the following actions for each possible character that it finds in format.

| Contents of format string | Conversion action taken |
|---|---|
| One or more white space characters | The function skips over the corresponding number of white space characters in the *monstr* money string (unless the formatting directive begins with the minus-sign modifier), up to the first character that is not white space or until no more characters can be scanned. White space characters are characters that the blank class of the LC_CTYPE category in the current locale defines. |
| A valid formatting directive | The function performs the specified conversion on the monetary element in the *monstr* money string. It replaces the formatting directive with an internal representation of the monetary element for conversion to the *money* value. Only one formatting directive is allowed in the *format* string. |
| Ordinary characters | The function must find the same ordinary character in the *monstr* money string. Any mismatch generates an error. The differing and subsequent characters in *monstr* remain unscanned.<br><br>You cannot include white space characters as ordinary characters. |

The formatting directive consists of the following sequence:

`%[flags][maximum_width]type_specifier`

| Argument | See |
|---|---|
| *flags* | "Field width" on page 4-22 |
| *maximum_width* | "Field width" on page 4-22 |
| *type_specifier* | "Valid type specifiers" on page 4-22 |

**Tip:** In the preceding format sequence, the square brackets indicate that the enclosed portion of the format is optional.

## Valid type specifiers

The type specifier is a letter or letters within a formatting directive that identify a format for the **ifx_gl_convert_money()** function to expect when it scans a monetary element of a *monstr* string. The LC_MONETARY locale-file category of the current locale might define these monetary formats. The **ifx_gl_convert_money()** function supports the following formatting directives to represent a monetary element.

**%i** Matches the international monetary format (which uses the **int_curr_symbol** subcategory of the LC_MONETARY category in the current locale) for the *monstr* argument. For example, in the default locale, the international monetary format for 1,234.56 is the "USD 1,234.56" string. Declare the corresponding *money* value as (double *).

**%n** Matches the national currency format (which uses the **currency_symbol** of the LC_MONETARY category in the current locale) for the *monstr* argument. For example, in the default locale, the national monetary format for 1,234.56 is the "$1,234.56" string. Declare the corresponding *money* value as (double *).

**%%** Matches a % character. No corresponding argument is needed.

If a formatting directive does not correspond to any of the preceding directives, the behavior of the conversion is undefined.

## Field width

You can specify an optional field width in a formatting directive. This information follows the initial percent symbol (%) of the formatting directive and has the following format:

`['][-][maximum_width]`

**[']** Indicates grouping rules.

If the specification begins with a single quotation mark ('), the field is expected to follow the grouping rules in the LC_MONETARY category of the current locale.

**[-]** Indicates field justification.

If the specification begins with a minus sign (-), the function expects the first character of the *monstr* money string to be the first character of the value to be converted. The function does not skip white space characters in *monstr* but reports them as an error.

If you omit the minus sign (-), the function assumes that the *monstr* string is right-aligned, and it skips any initial white space characters.

*maximum_width*
A decimal value that specifies the maximum number of characters to read from the *monstr* money string.

## Locale information

The LC_MONETARY category of the current locale affects the behavior of this function because it provides the locale-specific information for the scan of the *monstr* money string. For more information, see "The LC_MONETARY locale-file category" on page 3-2.

### Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_INVALIDFMT**
The *format* string is invalid.

**Related reference**:

"The ifx_gl_convert_number() function"

"The ifx_gl_format_money() function" on page 4-43

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_convert_number() function

The **ifx_gl_convert_number()** function converts a number string to its internal decimal representation.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_convert_number(number, numstr, format)
       mi_number *number;
       char *numstr;
       char *format;
```

*number*
A pointer to the variable that holds the internal decimal representation that **ifx_gl_convert_number()** creates from the *numstr* number string.

*numstr* A pointer to the first character of the number string that the function converts to its internal decimal representation.

*format* A pointer to the format string that determines how to interpret the *numstr* number string. For more information, see "Format string" on page 4-24

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_convert_number()** function converts the number string that *numstr* references to its internal decimal representation, which the function stores in the *number* argument. The function uses the format that the *format* string specifies to scan the *numstr* number string.

The *number* argument is a pointer to a **decimal (dec_t)** value.

The *number* argument is a pointer to an **mi_decimal** value.

## Format string

If *format* is NULL, the function uses the formats in the LC_NUMERIC category of the current GLS locale file to scan the *numstr* number string. If *format* is not NULL, it must point to a string that follows the rules that this section describes.

To convert the *numstr* number string, **ifx_gl_convert_number()** compares each character in *numstr* with the *format* string. It takes the following actions for each possible character that it finds in *format*.

| Contents of format string | Conversion action taken |
|---|---|
| One or more white space characters | The function skips over the corresponding number of white space characters in the *numstr* number string (unless the formatting directive begins with the minus-sign modifier), up to the first character that is not white space or until no more characters can be scanned. White space characters are characters that the blank class of the LC_CTYPE category in the current locale defines. |
| A valid formatting directive | The function performs the specified conversion on the numeric element in the *numstr* number string. It replaces the formatting directive with an internal representation of the numeric element for conversion to the *number* value. Only one formatting directive is allowed in the *format* string. |
| Ordinary characters | The function must find the same ordinary character in the *numstr* number string. Any mismatch generates an error. The differing and subsequent characters in *numstr* remain unscanned. |

The formatting directive consists of the following sequence:

```
%[flags][maximum_width[.minimum_width]]type_specifier
```

| Argument | See |
|---|---|
| *flags* | "Field width" on page 4-25 |
| *maximum_width* | "Field width" on page 4-25 |
| *minimum_width* | "Field width" on page 4-25 |
| *type_specifier* | "Valid type specifiers" |

**Tip:** In the preceding format sequence, the square brackets indicate that the enclosed portion of the format is optional.

## Valid type specifiers

The type specifier is a letter or letters within a formatting directive that identify a format for the **ifx_gl_convert_number()** function to expect when it scans a numeric element of a *numstr* string. These numeric formats are formats that the

LC_NUMERIC locale-file category of the current locale might define. The **ifx_gl_convert_number()** function supports the following formatting directives to represent a numeric element.

**%b** Matches a binary integer.

**%d** Matches a decimal integer

**%e** Matches a floating-point number

**%E** Same as **%e**.

**%f** Same as **%e**.

**%g** Same as **%e**.

**%G** Same as **%e**.

**%i** Same as **%d**.

**%o** Matches an octal integer.

**%q** Matches a base-4 integer.

**%u** Matches an unsigned decimal integer.

**%x** Matches a hexadecimal integer.

**%X** Same as **%x**.

## Field width

You can specify an optional field width in a formatting directive. This information follows the initial percent symbol (%) of the formatting directive and has the following format:

[-][*maximum_width*][.*minimum_width*]

**[-]** Indicates field justification.

If the specification begins with a minus sign (-), the function expects the first character of the *numstr* number string to be the first character of the value to be converted. The function does not skip white space characters in *numstr* but reports them as an error.

If you omit the minus sign (-), the function assumes that the *numstr* string is right-aligned, and it skips any initial white space characters.

*maximum_width*
A decimal value that specifies the maximum number of characters to read from the *numstr* money string.

*minimum_width*
Indicates an optional minimum field width. The minimum field width has the format of a period (.) followed by a decimal number. The *minimum_width* decimal number represents the minimum number of characters to read. If the function reads fewer than *minimum_width* characters, it generates an error (unless you specified left justification). When you specify left justification, the function skips any trailing white space to read the required number of characters.

## Locale information

The LC_NUMERIC category of the current locale affects the behavior of this function because it provides the locale-specific information for the scan of the *numstr* number string.

### Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_INVALIDFMT**
The *format* string is invalid.

**IFX_GL_PARAMERR**
The type specifier in the formatting directive is invalid.

**Related concepts**:

"The LC_NUMERIC locale-file category" on page 3-1

**Related reference**:

"The ifx_gl_convert_money() function" on page 4-20

"The ifx_gl_format_number() function" on page 4-47

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_cv_mconv() function

The **ifx_gl_cv_mconv()** function converts characters from one code set to another.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_cv_mconv(state, dst, dst_bytesleft, dst_codeset, src, src_bytesleft,
    src_codeset)
       conv_state_t *state;
       gl_mchar_t **dst;
       int *dst_bytesleft;
       char *dst_codeset;
       gl_mchar_t **src;
       int *src_bytesleft;
       char *src_codeset;
```

*state* Points to a **conv_state_t** structure. For more information, see "Convert fragmented strings" on page 4-27.

*dst* Points to a pointer to the first character in the destination buffer. If *dst* is NULL, the function updates **src*, **src_bytesleft*, and **dst_bytesleft*, but the converted data is not written to **dst*.

*dst_bytesleft*
Points to the maximum number of bytes to write to **dst*.

*dst_codeset*
A pointer to the name of the destination (target) code set.

*src* Points to a pointer to the first character in the source buffer.

*src_bytesleft*
Points to the number of bytes in **src* to convert. If *src_bytesleft* is NULL, the function converts data until it reaches a null character in the source buffer.

*src_codeset*
A pointer to the name of the source code

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_cv_mconv()** function converts the string of multibyte characters in **src* to the same characters, but encoded in another code set, and stores the result in the buffer that **dst* references. The **ifx_gl_cv_mconv()** function updates the following arguments:

- The function updates src to point to the byte that follows the last source character successfully converted.
- The function updates the integer that *src_bytesleft* references to the number of bytes in **src* that have not been converted. After a successful conversion, *src_bytesleft* points to 0.
- The function updates *dst* to point to the first character of the code-set-converted text.
- The function updates the integer that *dst_bytesleft* references to the number of bytes still available in **dst*.

## Specify code sets

The code sets, *src_codeset* and *dst_codeset*, can be any of the following items:

- Locale names
- Code-set names
- The IFX_GL_PROC_CS macro

For more information, see "Specify code-set names" on page 2-14.

## Calculate the size of the destination buffer

The number of bytes written to **dst* might be more or fewer than the number of bytes read from **src*. You can determine the number of bytes that will be written to **dst* in one of the following ways:

- The function **ifx_gl_cv_outbuflen()** calculates an estimate based on the **src_bytesleft* value.
- The following expression is the maximum number of bytes that will be written to **dst* for any value of **src* in any locale:

```
(*srcbytesleft) * IFX_GL_MB_MAX
```

Of these options, the last method is the fastest. The **ifx_gl_cv_outbuflen()** function is the slowest but the most precise.

## Convert fragmented strings

The state argument is a pointer to a **conv_state_t** structure. You must allocate a **conv_state_t** structure and set the **first_frag** and **last_frag** fields of the **conv_state_t** structure to tell the **ifx_gl_cv_mconv()** function whether the string to be converted is a fragment.

The following table lists the different fragments of a string and the corresponding values to which you must set these two **conv_state_t** fields.

| String fragment | Value of first_frag field | Value of last_frag field |
|---|---|---|
| String is the first of *n* fragments. | 1 | 0 |
| String is one of the 2nd, ..., *n*th-1 fragments. | 0 | 0 |
| String is the last (*n*th) fragment. | 0 | 1 |
| String is not fragmented; it is a complete string. | 1 | 1 |

**Important:** The **conv_state_t** structure contains other fields that are for internal use only. IBM Informix does not guarantee that these other internal fields of **conv_state_t** will not change in future releases. Therefore, to create portable code, set only the **first_frag** and **last_frag** fields of the **conv_state_t** structure.

## Locale information

For more information on the use of the **conv_state_t** structure, see "Preserve state information" on page 2-15.

## Return values

**0** The code-set conversion was successful, and the **src*, **src_bytesleft*, **dst*, and **dst_bytesleft* arguments have been updated. If the entire source buffer is converted, the **srcbytesleft* value is 0.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section. If the source conversion is stopped due to an error, the **srcbytesleft* value is greater than 0.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_FILEERR**
Retrieving the conversion information for the specified code sets failed. This failure might be due to invalid code-set names, a missing registry file, a missing code-set-conversion object file or one with an incorrect format, or insufficient memory for the code-set-conversion object.

**IFX_GL_EILSEQ**
The **src* value contains an invalid character. Conversion stops after both the last successfully converted source and destination character.

**IFX_GL_EINVAL**
The function cannot determine whether the last character of **src* is a valid character or the end of shift sequence because it would need to read more than **src_bytesleft* bytes from **src*. Conversion stops after the last successfully converted source and destination character.

**IFX_GL_E2BIG**
Not enough space is available in the destination buffer. Conversion stops after both the last successfully converted source and destination character.

**Related concepts**:

"Code-set conversion" on page 2-13

"Keep multibyte strings consistent" on page 2-26

"Fragment multibyte strings" on page 2-27

**Related reference**:

"The ifx_gl_conv_needed() function" on page 4-6

"The ifx_gl_cv_outbuflen() function"

"The ifx_gl_cv_sb2sb_table() function" on page 4-30

"The ifx_gl_lc_errno() function" on page 4-85

# The ifx_gl_cv_outbuflen() function

The **ifx_gl_cv_outbuflen()** function calculates an approximation of the number of bytes required to store a code-set converted multibyte character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_cv_outbuflen(dst_codeset, src_codeset, src_bytes)
       char *dst_codeset;
       char *src_codeset;
       int src_bytes;
```

*dst_codeset*
: A pointer to the name of the destination (target) code set.

*src_codeset*
: A pointer to the name of the source code set.

*src_bytes*
: The number of bytes in the buffer of multibyte characters to be code-set converted.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_cv_outbuflen()** function returns one of the following values:

- The exact number of bytes that a buffer of code-set-converted multibyte characters requires
- A close over-approximation of this number of bytes

Use this function to determine whether code-set conversion of multibyte characters can be performed in place. If the value that this function that returns is not equal to *src_bytes*, code-set conversion of multibyte characters cannot be performed in place. You must allocate a separate multibyte destination buffer. However, if the value that this function returns is exactly equal to *src_bytes*, you can perform multibyte code-set conversion in place.

The code sets, *src_codeset* and *dst_codeset*, can be any of the following:

- Locale names
- Code-set names
- The IFX_GL_PROC_CS macro

### Return values

**>=0** The number of bytes required to store multibyte characters of length *src_bytes* after they have been code-set converted.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_FILEERR**
Retrieval of the conversion information for the specified code sets failed. This failure might be due to invalid code-set names, a missing registry file, a missing code-set-conversion object file or one with an incorrect format, or insufficient memory for the code-set-conversion object.

**Related concepts**:

"Code-set conversion" on page 2-13

"Specify code-set names" on page 2-14

**Related reference**:

"The ifx_gl_conv_needed() function" on page 4-6

"The ifx_gl_cv_mconv() function" on page 4-26

"The ifx_gl_cv_sb2sb_table() function"

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_cv_sb2sb_table() function

The **ifx_gl_cv_sb2sb_table()** function returns the single-byte conversion table from the source code set to the destination code set.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_cv_sb2sb_table(dst_codeset, src_codeset, array)
       char *dst_codeset;
       char *src_codeset;
       unsigned char **array;
```

*dst_codeset*
A pointer to the name of the destination (target) code set.

*src_codeset*
A pointer to the name of the source code set.

*array* A pointer to a variable that points to a table of unsigned char values representing the conversion

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

If the code-set conversion from *src_codeset* to *dst_codeset* converts from one single-byte code set to another single-byte code set (where no substitution conversions occur), the **ifx_gl_cv_sb2sb_table()** function sets *array* to an array of

256 unsigned **char** values that represent the conversion. If the code-set conversion is not of this form, the function sets *array* to NULL.

The code sets, *src_codeset* and *dst_codeset*, can be any of the following:

- Locale names
- Code-set names
- The IFX_GL_PROC_CS macro

### Return values

**0** The function was successful, and the *array* argument points to the conversion table.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_FILEERR**
Retrieval of the conversion information for the specified code sets failed. This failure might be due to invalid code-set names, a missing registry file, a missing code-set-conversion object file or one with an incorrect format, or insufficient memory for the code-set-conversion object.

**Related concepts**:

"Code-set conversion" on page 2-13

"Specify code-set names" on page 2-14

**Related reference**:

"The ifx_gl_conv_needed() function" on page 4-6

"The ifx_gl_cv_mconv() function" on page 4-26

"The ifx_gl_cv_outbuflen() function" on page 4-29

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_format_date() function

The **ifx_gl_format_date()** function formats an internal date value to a date string.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_format_date(datestr, datebytes, date, format)
       char *datestr;
       int datebytes;
       mi_date *date;
       char *format;
```

*datestr* A pointer to the first character in the date string that the function formats from its internal *date* representation.

*datebytes*
The size of the *datestr* buffer for the date string. This size is the maximum size that the formatted string can reach.

*date* A pointer to the variable that holds the internal representation that **ifx_gl_format_date()** formats to create the *datestr* date string.

*format* A pointer to the format string that determines how to interpret the *datestr* date string. For more information, see "Format string."

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_format_date()** function uses the format that the *format* string specifies to format the internal date representation in *date* as a date string. It stores the resulting date string in a buffer that *datestr* references.

The *date* argument is a pointer to a **date (4-byte int)** value.

The *date* argument is a pointer to an **mi_date** value.

## Format string

If *format* is NULL, the **ifx_gl_format_date()** function determines the format for the *datestr* date string that it creates from the environment, as follows:

1. If the **DBDATE** environment variable is set, the function scans *datestr* according to the order of the format elements in **DBDATE**.
2. If the **GL_DATE** environment variable is set, the function scans *datestr* according to the specification of **GL_DATE**.
3. Otherwise, the function obtains the format from the **d_fmt** subcategory of the LC_TIME category in the current GLS locale file.

If *format* is not NULL, it must point to a string that follows the rules that this section describes. To format the *datestr* date string, **ifx_gl_format_date()** takes the following actions for each possible character that it finds in *format*.

| Contents of format string | Conversion action taken |
|---|---|
| Valid formatting directives | The function performs the specified formatting of the data in the *date* argument. It replaces the formatting directive with a string representation of a date element for the *datestr* date string. |
| Ordinary characters | The function copies the ordinary character unchanged to the *datestr* date string. Ordinary characters include the null terminator and white space characters. |

The formatting directive consists of the following sequence:

`%[modifiers][flag][width][.precision]type_specifier`

| Argument | See |
|---|---|
| *modifiers* | "Modified formatting directives" on page 4-34 |
| *flag* | "Field width and precision" on page 4-35 |
| *width* | "Field width and precision" on page 4-35 |
| *precision* | "Field width and precision" on page 4-35 |
| *type_specifier* | "Valid type specifiers" on page 4-33 |

**Tip:** In the preceding format sequence, the square brackets indicate that the enclosed portion of the format is optional.

## Valid type specifiers

The type specifier is a letter or letters within a formatting directive that identify a format for the **ifx_gl_format_date()** function to use when it creates a date element of a *datestr* string. These date formats are formats that the LC_TIME locale-file category of the current locale might define. The **ifx_gl_format_date()** function supports the following formatting directives to represent a date element.

**%a** Replaced by the abbreviated weekday name that the **abday** subcategory of LC_TIME defines.

**%A** Replaced by the full weekday name that the **day** subcategory of LC_TIME defines.

**%b** Replaced by the abbreviated month name that the **abmon** subcategory of LC_TIME defines.

**%B** Replaced by the full month name that the **mon** subcategory of LC_TIME defines.

**%C** Replaced by the century number (the year divided by 100 and truncated to an integer as a decimal number in the range 00-00).

**%d** Replaced by the day of the month as a decimal number in the range 1 - 31.

**%D** Same as **%m/%d/%y**.

**%e** Replaced by the day of the month as a decimal number in the range 1 - 31; a single digit is preceded by a space.

**%h** Same as **%b**.

**%j** Replaced by the day of the year as a decimal number in the range 1 - 366.

**%m** Replaced by the month as a decimal number in the range 1 - 12.

**%n** Replaced by a newline character.

**%t** Replaced by a tab character.

**%u** Replaced by the weekday as a decimal number in the range 1 - 7, with 1 representing Monday.

**%w** Replaced by the weekday as a decimal number in the range 0 - 6, with 0 representing Sunday.

**%x** Indicates use of the format that the **d_fmt** subcategory of LC_TIME defines.

**%y** Replaced by year without century as a decimal number in the range 0 -99.

**%Y** Replaced by year with century as a decimal number in the range 0 - 99.

**%%** Replaced by the % symbol.

If a formatting directive does not correspond to any of the preceding directives, the behavior of the conversion is undefined.

## Modified formatting directives

You can modify some formatting directives with format modifiers, which follow the percent symbol (%), to indicate use of an alternative format that the LC_TIME locale-file category of the current locale might define. The **ifx_gl_format_date()** function supports the following format modifiers.

**E** Use formats that include the era-based dates.

LC_TIME subcategory: **era**, **era_d_fmt**

**O** Use alternative locale-specific digits in dates.

LC_TIME subcategory: **alt_digits**

The alternative format replaces one that an unmodified formatting directive normally uses. If the alternative format does not exist for the current locale, the behavior is the same as if unmodified formatting directives were used. The **ifx_gl_format_date()** function supports the following modified formatting directives with the **E** and **O** modifier.

**%EC** Replaced by the name of the base year (period) that the **era** subcategory of LC_TIME defines.

**%Eg** Replaced by the abbreviated name of the base year (period) that the **era** subcategory of LC_TIME defines.

**%Ex** The value of the **era_d_fmt** subcategory of the LC_TIME category for the current locale is temporarily used as the format string.

**%Ey** Replaced by the offset from **%EC** (year only) that the **era** subcategory of LC_TIME defines.

**%EY** Replaced by the full alternative year representation that the **era** subcategory of LC_TIME defines.

**%Oe** Replaced by the day of the month, using the alternative digits that the **alt_digits** subcategory of LC_TIME defines. This formatting directive fills as needed with leading spaces.

**%Om** Replaced by the month, using the alternative digits that the **alt_digits** subcategory of LC_TIME defines.

**%Ou** Replaced by the weekday as a number, using the alternative digits that the **alt_digits** subcategory of LC_TIME defines (Monday=1).

**%Oy** Replaced by the value of **%Ey**, using the alternative digits that the **alt_digits** subcategory of LC_TIME defines.

The **ifx_gl_format_date()** function also supports the **i** modifier in the following formatting directives to support formats that are compatible with earlier IBM Informix date and time formatting.

**%iy** Replaced by the Informix **DBDATE** format Y2. This format prints the two-digit year offset.

**%iY** Replaced by the Informix **DBDATE** format Y4. This format prints the full four-digit year.

## Field width and precision

You can also specify an optional field width and precision in a formatting directive. This information precedes the type specifier in the formatting directive and has the following format:

`[- | 0][width][.precision]`

**[- | 0]**
Indicates field justification.

If the specification begins with a minus sign (-), the function left-aligns the field and pads it with spaces on the right. If the value begins with 0, the function right-aligns the field and pads it with zeros on the left. Otherwise, the function right-aligns the field and pads it with spaces on the left.

*width* A decimal value that specifies a minimum field width for the formatted *datestr* value.

*precision*
The format has the format of a period (.) followed by a decimal value. The field *width* specifier can optionally be followed by a *precision* directive. For more information about the purpose of *precision*, see the following table.

The values of *width* and *precision* affect each element of the directive. For example, **%6.4D** is interpreted as "**%6.4m/%6.4d/%6.4y**".

The precision value has the following effects on the different formatting directives.

| Formatting directives | Effect of precision |
|---|---|
| **%C**, **%d**, **%e**, **%Ey**, **%iy**, **%iY**, **%j**, **%m**, **%u**, **%w**, **%y**, **%Y** | The value of *precision* specifies the minimum number of digits to appear. If a directive supplies fewer digits than *precision* specifies, the function pads the value with leading zeros.<br><br>The **%d**, **%Ey**, **%iy**, **%m**, **%u**, **%w**, and **%y** directives have a default precision of 2. The **%j** directive has a default precision of 3. The **%Y** and **%iY** directives have a default precision of 4. |
| **%a**, **%A**, **%b**, **%B**, **%EC**, **%Eg**, **%h** | The value of *precision* specifies the maximum number of characters to be used. If a value to be formatted has more characters than the *precision* specifies, the function truncates the result on the right. |
| **%Ex**, **%EY**, **%n**, **%t**, **%x**, **%%** | The values of *width* and *precision* do not affect these directives. |
| Directives modified with **O** (alternative digits) | The field *width* relates to display width rather than actual number of digits; *precision* is still the minimum number of digits printed. |

## Locale information

The LC_TIME category of the current locale affects the behavior of this function because it provides the locale-specific information for the formatting of the *date* value.

## Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurs, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_E2BIG**
Formatting stopped due to lack of space in the buffer.

**IFX_GL_EBADF**
A formatting directive is invalid.

**IFX_GL_EDAYRANGE**
Day number is out of bounds.

**IFX_GL_EWKDAYRANGE**
Weekday number is out of bounds.

**IFX_GL_EYDAYRANGE**
Year day number is out of bounds.

**IFX_GL_EMONTHRANGE**
Month number is out of bounds.

**IFX_GL_EYEARRANGE**
Year number is out of bounds.

**IFX_GL_BADDAY**
Month (as a number) could not be scanned.

**IFX_GL_BADWKDAY**
Weekday (as a number) could not be scanned.

**IFX_GL_BADYDAY**
Day of year (as a number) could not be scanned.

**IFX_GL_BADMONTH**
Month could not be scanned.

**IFX_GL_BADYEAR**
Year could not be scanned.

**IFX_GL_BADERANAME**
Era name was not found.

**IFX_GL_BADERAOFFSET**
Era offset could not be scanned.

**Related concepts**:

"The LC_TIME locale-file category" on page 3-3

**Related reference**:

"The ifx_gl_convert_date() function" on page 4-7

"The ifx_gl_format_datetime() function"

"The ifx_gl_lc_errno() function" on page 4-85

# The ifx_gl_format_datetime() function

The **ifx_gl_format_datetime()** function formats an internal date/time value to a date/time string.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_format_datetime(datetimestr, datetimebytes, datetime, format)
       char *datetimestr;
       int datetimebytes;
       mi_datetime *datetime;
       char *format;
```

*datetimestr*
: A pointer to the first character in the date/time string that the function formats from its internal *datetime* representation.

*datetimebytes*
: The size of the *datetimestr* buffer for the date/time string. This size is the maximum size that the formatted string can reach.

*datetime*
: A pointer to the variable that holds the internal representation that **ifx_gl_format_datetime()** formats to create the *datetimestr* date/time string.

*format*
: A pointer to the format string that determines how to interpret the *datetimestr* date/time string. For more information, see "Format string."

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_format_datetime()** function uses the format that the *format* string specifies to format the internal date/time representation in *datetime* as a date/time string. It stores the resulting date/time string in a buffer that *datetimestr* references.

The *datetime* argument is a pointer to a **datetime (dtime_t)** value.

The *datetime* argument is a pointer to an **mi_datetime** value.

## Format string

If *format* is NULL, the **ifx_gl_format_datetime()** function determines the format of the *datetimestr* date/time string that it creates from the environment, as follows:

1. If the **DBTIME** environment variable is set, the function formats *datetimestr* according to**DBTIME**.
2. If the **GL_DATETIME** environment variable is set, the function formats *datetimestr* according to the specification of **GL_DATETIME**.

3. Otherwise, the function obtains the format from the **d_t_fmt** subcategory of the LC_TIME category in the current GLS locale file.

If *format* is not NULL, it must point to a string that follows the rules that this section describes. To format the *datetimestr* date/time string, **ifx_gl_format_datetime()** takes the following actions for each possible character that it finds in *format*.

| Contents of format string | Conversion action taken |
|---|---|
| Valid formatting directives | The function performs the specified formatting of the data in the *datetime* argument. It replaces the formatting directive with a string representation of a date or time element for the *datetimestr* date/time string. |
| Ordinary characters | The function copies the ordinary character unchanged to the *datetimestr* date/time string. Ordinary characters include the null terminator and white space characters. |

The formatting directive consists of the following sequence:
`%[modifiers][flag][width][.precision]type_specifier`

| Argument | See |
|---|---|
| *modifiers* | "Modified formatting directives" on page 4-40 |
| *flag* | "Field width and precision" on page 4-41 |
| *width* | "Field width and precision" on page 4-41 |
| *precision* | "Field width and precision" on page 4-41 |
| *type_specifier* | "Valid type specifiers" |

**Tip:** In the preceding format sequence, the square brackets indicate that the enclosed portion of the format is optional.

## Valid type specifiers

The type specifier is a letter or letters within a formatting directive that identify a format for the **ifx_gl_format_datetime()** function to use when it creates a date or time element of a *datetimestr* string. These date and time formats are formats that the LC_TIME locale-file category of the current locale might define. The **ifx_gl_format_datetime()** function supports the following formatting directives to represent a date or time element that the LC_TIME locale-file category of the current locale might define.

**%a** Replaced by the abbreviated weekday name that the **abday** subcategory of LC_TIME defines.

**%A** Replaced by the full weekday name that the **day** subcategory of LC_TIME defines.

**%b** Replaced by the abbreviated month name that the **abmon** subcategory of LC_TIME defines.

**%B** Replaced by the full month name that the **mon** subcategory of LC_TIME defines.

**%c** Indicates use of the format that the **d_t_fmt** subcategory of LC_TIME defines.

**%C** Replaced by the century number (the year divided by 100 and truncated to an integer as a decimal number [00-00]).

**%d** Replaced by the day of the month as a decimal number in the range 1 - 31.

**%D** Same as **%m/%d/%y**.

**%e** Replaced by the day of the month as a decimal number in the range 1 - 31; a single digit is preceded by a space.

**%F[***n***]** Replaced by the microsecond as a decimal number. An optional precision specification, *n*, can follow **F**. This *n* value must be 1 - 5.

**%h** Same as **%b**.

**%H** Replaced by the hour (24-hour clock) as a decimal number in the range 0 - 23.

**%I** Replaced by the hour (12-hour clock) as a decimal number in the range 0 -12.

**%j** Replaced by the day of the year as a decimal number in the range 1 - 366.

**%m** Replaced by the month as a decimal number in the range 1 -12.

**%M** Replaced by the minute as a decimal number in the range 0- 59.

**%n** Replaced by a newline character.

**%p** Replaced by the locale equivalent of either a.m. or p.m. that the **am_pm** subcategory of LC_TIME defines.

**%r** Indicates use of the format that the **t_fmt_amp** subcategory of LC_TIME defines.

**%R** Replaced by the time in 24-hour notation (**%H:%M**).

**%S** Replaced by the second as a decimal number.

**%t** Replaced by a tab character.

**%T** Replaced by the time (**%H:%M:%S**).

**%u** Replaced by the weekday as a decimal number in the range 1 - 7, with 1 representing Monday.

**%w** Replaced by the weekday as a decimal number in the range 0 - 6, with 0 representing Sunday.

**%x** Indicates use of the format that the **d_fmt** subcategory of LC_TIME defines.

**%X** Indicates use of the format that the **t_fmt** subcategory of LC_TIME defines.

**%y** Replaced by year without century as a decimal number in the range 0 - 99.

**%Y** Replaced by year with century as a decimal number 0 - 99.

**%%** Replaced by the % symbol.

If a formatting directive does not correspond to any of the preceding directives, the behavior of the conversion is undefined.

## Modified formatting directives

You can modify some formatting directives with format modifiers, which follow the percent symbol (%), to indicate use of an alternative format that the LC_TIME locale-file category of the current locale might define. The **ifx_gl_format_datetime()** function supports the following format modifiers.

**E** Use formats that include the era-based dates and times.

LC_TIME subcategory: **era**, **era_d_fmt**, **era_t_fmt**, **era_d_t_fmt**

**O** Use alternative locale-specific digits in dates and times.

LC_TIME subcategory: **alt_digits**

The alternative format replaces one that an unmodified formatting directive normally uses. If the alternative format does not exist for the current locale, the behavior is the same as if unmodified formatting directives were used. The **ifx_gl_format_datetime()** function supports the following modified formatting directives with the **E** and **O** modifier.

**%Ec** Indicates use of the format that the **era_d_t_fmt** subcategory of LC_TIME defines.

**%EC** Replaced by the name of the base year (period) that the **era** subcategory of LC_TIME defines.

**%Eg** Replaced by the abbreviated name of the base year (period) in the alternative representation that the **era** subcategory of LC_TIME defines.

**%Ex** Indicates use of the format that the **era_d_fmt** subcategory of LC_TIME defines.

**%EX** Indicates use of the format that the **era_t_fmt** subcategory of LC_TIME defines.

**%Ey** Replaced by the offset from **%EC** (year only) that the **era** subcategory of LC_TIME defines.

**%EY** Replaced by the full alternative year representation that the **era** subcategory of LC_TIME defines.

**%Oe** Replaced by the day of the month, using the alternative digits that the **alt_digits** subcategory of LC_TIME defines. This formatting directive fills as needed with leading spaces.

**%OH** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to format the hour (24-hour clock).

**%OI** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to format the hour (12-hour clock).

**%Om** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to format the month.

**%OM** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to format the minutes.

**%OS** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to format the seconds.

**%Ou** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to format the weekday as a number (Monday=1).

**%Oy** Uses the alternative digits that the **alt_digits** subcategory of LC_TIME defines to format the value of %Ey.

The **ifx_gl_format_datetime()** function also supports the **i** modifier in the following formatting directives to support formats that are compatible with earlier IBM Informix date and time formatting.

**%iF[***n***]** Replaced by the Informix **DBTIME** format **%F**. An optional precision specification, *n*, can follow **F**. This n value must be 1 - 5.

**%iy** Replaced by the Informix **DBDATE** format Y2. This format prints the two-digit year offset.

**%iY** Replaced by the Informix **DBDATE** format Y4. This format prints the full four-digit year.

## Field width and precision

You can also specify an optional field width and precision in a formatting directive. This information precedes the type specifier in the formatting directive and has the following format:

```
[- | 0][width][.precision]
```

**[- | 0]**
Indicates field justification.

If the specification begins with a minus sign (-), the function left-aligns the field and pads it with spaces on the right. If the value begins with 0, the function right-aligns the field and pads it with zeros on the left. Otherwise, the function right-aligns the field and pads it with spaces on the left.

*width* A decimal value that specifies a minimum field width for the formatted *datetimestr* value.

*precision*
The format has the format of a period (.) followed by a decimal value. The field *width* specifier can optionally be followed by a *precision* directive. For more information about the purpose of *precision*, see the following table.

The values of *width* and *precision* affect each element of the **%D**, **%R**, and **%T** directives. For example, **%6.4D** is interpreted as "**%6.4m/%6.4d/%6.4y**".

The precision value has the following effects on the different formatting directives.

| Formatting directives | Effect of precision |
|---|---|
| **%C**, **%d**, **%e**, **%Ey**, **%F**, **%H**, **%iF**, **%iy**, **%iY**, **%I**, **%j**, **%m**, **%M**, **%S**, **%u**, **%w**, **%y**, **%Y** | The value of *precision* specifies the minimum number of digits to appear. If a directive supplies fewer digits than *precision* specifies, the function pads the value with leading zeros.<br><br>The **%d**, **%Ey**, **%H**, **%iF**, **%iy**, **%m**, **%M**, **%S**, **%u**, **%w**, and **%y** directives have a default precision of 2. The **%j** directive has a default precision of 3. The **%Y** and **%iY** directives have a default precision of 4. |

| Formatting directives | Effect of precision |
|---|---|
| **%a**, **%A**, **%b**, **%B**, **%EC**, **%Eg**, **%h**, **%p** | The value of *precision* specifies the maximum number of characters to be used. If a value to be formatted has more characters than the *precision* specifies, the function truncates the result on the right. |
| **%c**, **%Ec**, **%Ex**, **%EX**, **%EY**, **%n**, **%r**, **%t**, **%x**, **%X**, **%%** | The values of *width* and *precision* do not affect these directives. |
| Directives modified with **O** (alternative digits) | The field *width* relates to display width rather than actual number of digits; *precision* is still the minimum number of digits printed. |

## Locale information

The LC_TIME category of the current locale affects the behavior of this function because it provides the locale-specific information for the formatting of the *datetime* value.

## Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurs, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_E2BIG**
Formatting stopped due to lack of space in the buffer.

**IFX_GL_EBADF**
A formatting directive is invalid.

**IFX_GL_EFRACRANGE**
Fraction of second is out of bounds.

**IFX_GL_ESECONDRANGE**
Second is out of bounds.

**IFX_GL_EHOURRANGE**
Hour is out of bounds.

**IFX_GL_EMINUTERANGE**
Minute is out of bounds.

**IFX_GL_EDAYRANGE**
Day number is out of bounds.

**IFX_GL_EWKDAYRANGE**
Weekday number is out of bounds.

**IFX_GL_EYDAYRANGE**
Year day number is out of bounds.

**IFX_GL_EMONTHRANGE**
Month number is out of bounds.

**IFX_GL_EYEARRANGE**
Year number is out of bounds.

**IFX_GL_EERAOFFRANGE**
Era offset is out of bounds.

**IFX_GL_BADFRAC**
Fraction could not be scanned.

**IFX_GL_BADSECOND**
Second could not be scanned.

**IFX_GL_BADMINUTE**
Minute could not be scanned.

**IFX_GL_BADHOUR**
Hour could not be scanned.

**IFX_GL_BADDAY**
Month (as a number) could not be scanned.

**IFX_GL_BADWKDAY**
Weekday (as a number) could not be scanned.

**IFX_GL_BADYDAY**
Day of year (as a number) could not be scanned.

**IFX_GL_BADMONTH**
Month could not be scanned.

**IFX_GL_BADYEAR**
Year could not be scanned.

**IFX_GL_BADERANAME**
Era name was not found.

**IFX_GL_BADERAOFFSET**
Era offset could not be scanned.

**Related concepts**:

"The LC_TIME locale-file category" on page 3-3

**Related reference**:

"The ifx_gl_convert_datetime() function" on page 4-13

"The ifx_gl_format_date() function" on page 4-31

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_format_money() function

The **ifx_gl_format_money()** function formats an internal money value to a money string.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_format_money(monstr, monstrbytes, money, format)
       char *monstr;
       int monstrbytes;
       mi_money *money;
       char *format;
```

*monstr* A pointer to the first character of the money string that the function formats to its internal *money* representation.

*monstrbytes*
The size of the *monstr* buffer for the money string. This size is the maximum size that the formatted string can reach.

*money* A pointer to the variable that holds the internal representation that **ifx_gl_format_money()** formats to create the *monstr* money string.

*format* A pointer to the format string that determines how to format the *monstr* money string. For more information, see "Format string"

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_format_money()** function uses the format that the *format* string specifies to format the internal money representation in *money* as a money string. It stores the resulting money string in a buffer that *monstr* references.

The *money* argument is a pointer to a **decimal (dec_t)** value.

The *money* argument is a pointer to an **mi_money** value.

## Format string

If *format* is NULL, the **ifx_gl_format_money()** function determines the format of the *monstr* money string that it creates from the environment, as follows:

1. If the **DBMONEY** environment variable is set, the function formats *monstr* according to **DBMONEY**.
2. Otherwise, the function uses the monetary formats in the LC_MONETARY category of the current GLS locale.

If *format* is not NULL, it must point to a string that follows the rules that this section describes. To format the *monstr* money string, the function **ifx_gl_format_money()** takes the following actions for each possible character that it finds in format.

| Contents of format string | Conversion action taken |
|---|---|
| A valid formatting directive | The function performs the specified formatting of the data in the *money* argument. It replaces the formatting directive with a string representation of a money element for the *monstr* money string. Only one formatting directive is allowed in the *format* string. |
| Ordinary characters | The function copies the ordinary character unchanged to the *monstr* money string. Ordinary characters include the null terminator and white space characters. |

The formatting directive consists of the following sequence:

```
%[modifiers][flag][width[#left_precision][.right_precision]]type_specifier
```

| Argument | See |
|---|---|
| *modifiers* | "Modified formatting directives" |
| *flag* | "Field width and precision" on page 4-46 |
| *width* | "Field width and precision" on page 4-46 |
| *left_precision* | "Field width and precision" on page 4-46 |
| *right_precision* | "Field width and precision" on page 4-46 |
| *type_specifier* | "Valid type specifiers" |

**Tip:** In the preceding format sequence, the square brackets indicate that the enclosed portion of the format is optional.

## Valid type specifiers

The type specifier is a letter or letters within a formatting directive that identify a format for the **ifx_gl_format_money()** function to use when it creates a monetary element of a *monstr* string. These monetary formats are formats that the LC_MONETARY locale-file category of the current locale might define. The **ifx_gl_format_money()** function supports the following formatting directives.

**%i** Replaced by the international monetary format (which uses the **int_curr_symbol** subcategory of the LC_MONETARY category in the current locale) for the *money* argument. For example, in the default locale, the international monetary format for 1,234.56 is `USD 1,234.56`.

**%n** Replaced by the national currency format (which uses the **currency_symbol** of the LC_MONETARY category in the current locale) for the *money* argument. For example, in the default locale, the national monetary format for 1,234.56 is `$1,234.56`.

**%%** Replaced by the % symbol.

If a formatting directive does not correspond to any of the preceding directives, the behavior of the conversion is undefined.

## Modified formatting directives

You can modify some formatting directives with format modifiers, which follow the percent symbol (%), to indicate use of an alternative format. The **ifx_gl_format_money()** function supports the following format modifiers.

**=***f* Use the single-byte character *f* as the numeric-fill character. The fill character must be representable in a single byte to work with precision and width counts. The default numeric-fill character is the blank (space) character. This flag does not affect field-width filling, which always uses the space character. Unless you specify *left_precision*, this flag is ignored.

**^** Do not format the monetary amount with thousands separators. The default action is to insert the thousands separators if the LC_MONETARY category of the current locale defines them.

**+ or (** Specify how to represent positive and negative monetary amounts. You can specify only + or (. If you specify the plus sign (+) , the function uses the equivalent of + and - that the LC_MONETARY category of the current locale defines. For example, in the default locale, a plus sign means an empty string for positive values and a minus sign (-) for negative values. If

you do not specify either flag, the function uses the locale equivalent of the minus sign (-) for negative values and no sign for positive values.

**!** Suppress the currency symbol from the formatted result.

## Field width and precision

You can also specify an optional field width and precision in a formatting directive. This information precedes the type specifier in the formatting directive and has the following format:

`[-][width[#left_precision][.right_precision]]`

**-** Indicates field justification. If the specification begins with a minus sign (-), the function left-aligns the field and pads it with spaces on the right. Otherwise, the function right-aligns the field and pads it with spaces on the left.

*width* A decimal value that specifies a minimum field width, in characters, in which the resulting string is right-aligned. The default field width is 0.

*left_precision*
A string that specifies the maximum number of digits expected to the left of the decimal separator. This option causes an amount to be formatted as if it has the number of digits that *n* specifies. If more than *n* digit positions are required, the function ignores this precision specification.

*right_precision*
A decimal value that specifies the number of digits after the decimal separator. If the value of *right_precision* is 0, no decimal separator appears in the formatted string. If you do not include *right_precision*, the function uses a default, which the current locale specifies. The amount that is being formatted is rounded to the specified number of digits before formatting.

You can use this *left_precision* option to:

- keep the formatted output from multiple calls to this function aligned in the same columns.
- fill unused positions with a special character, such as the asterisk character in the "$***123.45" string.

For *left_precision*, digit positions in excess of those positions required are filled with the numeric-fill character (see the description of the *=f* modifier in "Modified formatting directives" on page 4-45). If grouping has not been suppressed with the ^ modifier, and it is defined for the current locale, the function inserts thousands separators before the fill characters (if any) are added. Thousands separators are not applied to fill characters even if the fill character is a digit.

To ensure alignment, any characters that appear before or after the number in the formatted output, such as currency or sign symbols, are padded as necessary with space characters to make their positive and negative formats an equal length.

## Locale information

The LC_MONETARY category of the current locale affects the behavior of this function because it provides the locale-specific information for the formatting of the *money* value. For more information, see "The LC_MONETARY locale-file category" on page 3-2.

### Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following value.

**IFX_GL_ENOSYS**
The formatting directive is not supported.

**IFX_GL_E2BIG**
Formatting stopped due to insufficient space in the buffer.

**Related concepts**:

"The LC_MONETARY locale-file category" on page 3-2

**Related reference**:

"The ifx_gl_convert_money() function" on page 4-20

"The ifx_gl_format_number() function"

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_format_number() function

The **ifx_gl_format_number()** function formats an internal decimal value to a number string.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_format_number(numstr, numstrbytes, number, format)
       char *numstr;
       int numstrbytes;
       mi_decimal *number;
       char *format;
```

*numstr* A pointer to the first character in the number string that the function formats from its internal *number* representation.

*numstrbytes*
The size of the *numstr* buffer for the number string. This size is the maximum size that the formatted string can reach.

*number*
A pointer to the variable that holds the internal representation that **ifx_gl_format_number()** formats to create the *numstr* number string.

*format* A pointer to the format string that determines how to format the *numstr* number string. For more information, see "Format string" on page 4-48

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_format_number()** function uses the format that the *format* string specifies to format the internal decimal representation in *number* as a number string. It stores the resulting number string in a buffer that *numstr* references.

The *number* argument is a pointer to a **decimal (dec_t)** value.

The *number* argument is a pointer to an **mi_decimal** value.

## Format string

If *format* is NULL, the function uses the numeric formats in the LC_NUMERIC category of the current GLS locale file to format the *numstr* number string that it creates.

If *format* is not NULL, it must point to a string that follows the rules that this section describes. To format the *numstr* number string, **ifx_gl_format_number()** takes the following actions for each possible character that it finds in *format*.

| Contents of format string | Conversion action taken |
|---|---|
| A valid formatting directive | The function performs the specified formatting of the data in the *number* argument. It replaces the formatting directive with a string representation of a numeric element for the *numstr* number string. Only one formatting directive is allowed in the *format* string. |
| Ordinary characters | The function copies the ordinary character unchanged to the *numstr* number string. Ordinary characters include the null terminator and white space characters. |

The formatting directive consists of the following sequence:
`%[modifiers][flags][width[.precision]]type_specifier`

| Argument | See |
|---|---|
| *modifiers* | "Modified formatting directives" on page 4-49 |
| *flags* | "Field width and precision" on page 4-50 |
| *width* | "Field width and precision" on page 4-50 |
| *precision* | "Field width and precision" on page 4-50 |
| *type_specifier* | "Valid type specifiers" |

**Tip:** In the preceding format sequence, the square brackets indicate that the enclosed portion of the format is optional.

## Valid type specifiers

The type specifier is a letter or letters within a formatting directive that identify a format for the **ifx_gl_format_number()** function to use when it creates a numeric element of a *numstr* string. These numeric formats are formats that the

LC_NUMERIC locale-file category of the current locale might define. The **ifx_gl_format_number()** function supports the following formatting directives.

**%b** Replaced by the binary representation of *number*.

**%d** Replaced by the decimal representation of *number*.

**%e** Replaced by the style [-]*d.ddd*e[+|-]*dd* of number (where *d* is a digit).

**%E** Replaced by the style [-]*d.ddd*E[+|-]*dd* of number (where *d* is a digit).

**%f** Replaced by the style [-]*ddd.ddd* of number (where *d* is a digit). The default precision is 6.

**%g** Replaced by the style of **%f** or **%e**. The **%e** directive is used only if the converted exponent is less than -4 or greater than or equal to the precision. The precision specifies the number of significant digits. Trailing zeros are removed from the fractional portion of the result.

**%G** Same as **%g** , except for the replacement of **%E** for **%e**.

**%i** Same as **%d**.

**%o** Replaced by the octal representation of *number*.

**%q** Replaced by the base-4 representation of *number*.

**%u** Replaced by the unsigned decimal representation of *number*.

**%x** Replaced by the hexadecimal representation of *number*, using the hexadecimal characters `a` through `f`.

**%X** Same as **%x**, but use uppercase hexadecimal characters (`A` through `F`) instead of lowercase letters (`a` through `f`).

If a formatting directive does not correspond to any of the preceding directives, the behavior of the formatting is undefined.

## Modified formatting directives

You can modify some formatting directives with format modifiers, which follow the percent symbol (%), to indicate use of an alternative format. The **ifx_gl_format_number()** function supports the following format modifiers.

**'** Separates the significant digits of the converted number with the grouping character defined in the locale, according to the grouping format also defined in the locale.

**+** Begins the result of a signed conversion with a positive or negative sign, which the locale defines.

**-** Left-justifies the result of the conversion within the field. space Prefixes the result with a space character (unless the + modifier exists) if the beginning of a signed conversion is not a sign.

**#** Converts the value to an alternate form. For a **%o** directive, the first digit of the result is forced to be 0. For a **%x** (or **%X**) directive, a nonzero result will have a leading "0x" (or "0X"). For a **%e**, **%E**, **%f**, **%g**, or **%G** directive, the result always has a decimal-separator character.

**0** Pads the field width with leading zeros (unless the - modifier exists).

If the alternative format does not exist for the current locale, the function uses the unmodified formatting directive.

## Field width and precision

You can also specify an optional field width and precision in a formatting directive. This information precedes the type specifier in the formatting directive and has the following format:

`[- | 0][width[.precision]]`

**[- | 0]**
Indicates field justification. If the specification begins with a minus sign (-), the function left-aligns the field and pads it with spaces on the right. If the value begins with 0, the function right-aligns the field and pads it with zeros on the left. Otherwise, the function right-aligns the field and pads it with spaces on the left.

*width* A decimal value that specifies a minimum field width for the formatted string.

*precision*
The format has the format of a period (.) followed by a decimal value. The field width specifier can optionally be followed by a *precision* directive.

For the **%d**, **%o**, **%u**, **%x**, and **%X** directives, the value of *precision* specifies the minimum number of digits to appear. If a directive supplies fewer digits than specified by the precision, it is padded with leading zeros.

For the **%e**, **%E**, and **%f** directives, the value of *precision* specifies the number of digits to appear after the decimal separator.

For the **%g**, and **%G** directives, the value of *precision* specifies the maximum number of significant digits.

## Locale information

The LC_NUMERIC category of the current locale affects the behavior of this function because it provides the locale-specific information for the formatting of the *number* value.

## Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_INVALIDFMT**
The *format* string is invalid.

**IFX_GL_PARAMERR**
The type specifier in the formatting directive is invalid.

**IFX_GL_E2BIG**
Formatting stopped due to lack of space in the buffer.

**Related concepts**:

"The LC_NUMERIC locale-file category" on page 3-1

**Related reference**:

"The ifx_gl_convert_number() function" on page 4-23

"The ifx_gl_format_money() function" on page 4-43

"The ifx_gl_lc_errno() function" on page 4-85

# The ifx_gl_getmb() function

The **ifx_gl_getmb()** function obtains a single multibyte character from a user-specified location.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_fgetmb(mb, funcp, v, bytes_got)
       gl_mchar_t *mb;
       int (*funcp)(void *v);
       void *v;
       int *bytes_got;
```

*mb* A pointer to the multibyte character whose bytes the *funcp* function reads from a specified location.

*funcp* A pointer to a function that you define to specify the location from which to read the multibyte character.

*v* A pointer that the **ifx_gl_getmb()** function passes to the *funcp* function each time that it is called.

*bytes_got*
A pointer to an integer that **ifx_gl_getmb()** sets to indicate the number of bytes that the *funcp* function has successfully read.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_getmb()** function calls a function that you define to obtain the bytes that form one multibyte character from a specified location. This multibyte character is then written to the *mb* buffer.

The *funcp* argument is a pointer to a function that you must define as follows:

```
int funcp(void *v)
```

The pointer *v* is passed to *funcp* each time that it is called. On success, *funcp* must return a value 0 - 255, inclusive, which is the next byte of the multibyte character. On failure, *funcp* must return -1.

This function calls *funcp* until one of the following occurs:

- A complete multibyte character has been formed.
- The *funcp* function returns a byte that forms an illegal character when appended to the bytes already read.
- The *funcp* function fails.

All bytes obtained by calling *funcp* are guaranteed to be written to mb. This function assumes that mb is large enough to hold the result. You can determine what will be written to mb in either of the following ways:

- The function **ifx_gl_mb_loc_max()** calculates the maximum number of bytes in any multibyte character for the current locale.
- The macro IFX_GL_MB_MAX is the maximum number of bytes in any multibyte character in any locale. This value is always equal to or greater than the value that **ifx_gl_mb_loc_max()** returns.

Of these two options, the macro IFX_GL_MB_MAX is faster, and it can be used to initialize static buffers. The function **ifx_gl_mb_loc_max()** is slower but more precise.

The number of bytes that *funcp* successfully reads is returned in *bytes_got* (even when *funcp* fails).

### Return values

**0 - 255** The value of the next byte for the multibyte character

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The bytes read from *funcp* do not form a valid multibyte character.

**IFX_GL_EINVAL**
The *funcp* function returned -1 in the middle of an otherwise valid multibyte character.

**IFX_GL_EOF**
The *funcp* function returned -1 on the first call.

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mb_loc_max() function" on page 4-86

"The ifx_gl_putmb() function" on page 4-113

## The ifx_gl_init() function

The **ifx_gl_init()** function initializes the current processing locale for the current program.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_init(void)
```

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Not needed |

## Usage

The **ifx_gl_init()** function initializes a global locale structure that identifies the current processing locale. Because this locale structure is global, the IBM Informix GLS functions do not require the current locale as an argument. Any program that establishes its own connections must call the **ifx_gl_init()** function before it calls any other Informix GLS functions.

ESQL/C applications establish their own connections to a database server and therefore must call the **ifx_gl_init()** function before any Informix GLS functions to establish a current processing locale.

DataBlade client applications establish their own connections to a database server and therefore must call the **ifx_gl_init()** function before any Informix GLS functions to establish a current processing locale.

However, a DataBlade UDR executes in the context of an established connection and therefore has an established current processing locale. DataBlade UDRs are not required to call the **ifx_gl_init()** function.

For both ESQL/C and DataBlade client applications, **ifx_gl_init()** initializes the current processing locale to the client locale.

## Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either specifiers for locale or code-set conversion (such as GLS environment variables) are incorrect, or the code-set name registry file could not be found.

**IFX_GL_ENOMEM**
Not enough memory is available to allocate a new locale object or a new code-set-conversion object.

**IFX_GL_FILEERR**
The locale or code-set conversion file could not be found, is not readable, or has the wrong format.

**IFX_GL_INVALIDLOC**
An attempt to create a locale with incompatible code sets occurred.

**IFX_GL_ELOCTOOWIDE**
The locale contains characters that are wider than the library allows.

**IFX_GL_BADOBJVER**
The locale object version is not compatible with the current library.

**IFX_GL_BADFILEFORM**
Bad format was found in the code-set registry file.

**Related concepts**:

"Initialize the Informix GLS library" on page 1-8

# The ifx_gl_ismalnum() function

The **ifx_gl_ismalnum()** function determines whether a multibyte character contains an alphabetic or digit character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismalnum(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismalnum()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismalnum()** function tests whether *mb* is in the alnum character class (either the alpha or digit class) according to the rules of the current locale. The alpha class contains all characters from the upper and lower classes. The digit class contains only the 10 ASCII digit characters: '0' to '9' (ASCII 0x030 to 0x039).

To determine whether a multibyte character is defined only in the alpha class, use the **ifx_gl_ismalpha()** function. To determine whether a multibyte character is defined only in the digit class, use the **ifx_gl_ismdigit()** function.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the alnum character class.

## Return values

**>0** The **mb* character is in the alpha or digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismalpha() function"
"The ifx_gl_ismdigit() function" on page 4-59
"The ifx_gl_ismlower() function" on page 4-62
"The ifx_gl_ismupper() function" on page 4-68
"The ifx_gl_iswalnum() function" on page 4-71
"The ifx_gl_lc_errno() function" on page 4-85

# The ifx_gl_ismalpha() function

The **ifx_gl_ismalpha()** function determines whether a multibyte character contains an alphabetic character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismalpha(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismalpha()** function reads as many bytes as necessary from *mb* to form a complete character. For more information about *mb_byte_limit*, see "Multibyte-character termination" on page 2-23.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismalpha()** function tests whether *mb* is in the alpha character class according to the rules of the current locale. The alpha class contains all characters from the upper and lower classes. In addition to the uppercase and lowercase Latin-based characters, this class includes any alphabetic characters that the locale might define, including:

- Asian ideographic characters; for example, Kanji characters.
- Asian phonetic characters; for example, single-byte and multibyte Katakana and Hiragana characters.
- non-ASCII digit characters (see the digit class).

- Latin-based alphabetic characters that do not have a case-equivalent character.
- user-defined characters.
- vendor-defined characters.

Characters in the alpha class are also in the graph and print classes. No characters in the digit, blank, space, punct, or cntrl classes are in the alpha class.

To determine the case of an alphabetic multibyte character, you can use the **ifx_gl_ismupper()** and **ifx_gl_ismlower()**functions. Use the function **ifx_gl_ismalnum()** to test whether a multibyte character is an alphabetic character or a digit.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the alpha character class.

## Return values

**>0** The **mb* character is in the alpha character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error. For more information about this error, see "Keep multibyte strings consistent" on page 2-26.

**Related concepts**:

"Character classification" on page 2-5

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismalnum() function" on page 4-54

"The ifx_gl_ismgraph() function" on page 4-61

"The ifx_gl_ismlower() function" on page 4-62

"The ifx_gl_ismprint() function" on page 4-64

"The ifx_gl_ismupper() function" on page 4-68

"The ifx_gl_iswalpha() function" on page 4-72

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_tomlower() function" on page 4-114

"The ifx_gl_tomupper() function" on page 4-116

# The ifx_gl_ismblank() function

The **ifx_gl_ismblank()** function determines whether a multibyte character contains a horizontal-space character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismblank(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismblank()** function reads as many bytes as necessary from *mb* to form a complete character. For more information about *mb_byte_limit*, see "Multibyte-character termination" on page 2-23.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismblank()** function tests whether *mb* is in the blank character class according to the rules of the current locale. The blank character class includes the single-byte space (ASCII 0x020) and tab character (ASCII 0x009, ^I) plus any multibyte version of these characters that the locale defines. Characters in the blank class are also in the space class. No characters in the upper, lower, alpha, digit, xdigit, punct, or graph classes are in the blank class.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the blank character class.

### Return values

**>0** The **mb* character is in the blank character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismspace() function" on page 4-67

"The ifx_gl_iswblank() function" on page 4-74

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_ismcntrl() function

The **ifx_gl_ismcntrl()** function determines whether a multibyte character contains a control character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismcntrl(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismcntrl()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_ismcntrl()** function tests whether *mb* is in the cntrl character class according to the rules of the current locale. The cntrl character class contains the

single-byte control characters: alert, backspace, tab, newline, vertical tab, form feed, carriage return, NUL, SOH, STX, ETX, EOT, ENQ, ACK, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, IS4, IS3, IS2, IS1, and DEL. These characters are the ASCII characters whose code points are in the range 0x00 to 0x1F. In addition, the cntrl class contains any other control characters that the locale might define.

No characters in the upper, lower, alpha, digit, xdigit, punct, graph, or print classes are in the cntrl class.

### Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the cntrl character class.

### Return values

**>0** The **mb* character is in the cntrl character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5

"Multibyte-character termination" on page 2-23

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_iswcntrl() function" on page 4-75

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_ismdigit() function

The **ifx_gl_ismdigit()** function determines whether a multibyte character contains a decimal digit.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismdigit(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismdigit()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismdigit()** function tests whether *mb* is in the digit character class according to the rules of the current locale. The digit character class contains only the 10 ASCII digit characters (ASCII 0x030 to 0x039). Any multibyte versions or alternative representations of these digits that the locale might define (for example, Hindi or Kanji digits) are not in this class. Instead, they are in the alpha class.

Characters in this class are also in the xdigit, graph, and print classes. No characters in the upper, lower, alpha, blank, space, punct, or cntrl classes are in the digit class.

To determine whether a multibyte character contains a hexadecimal digit, you can use the **ifx_gl_ismxdigit()** function. Use the **ifx_gl_ismalnum()** function to test whether a multibyte character is an alphabetic character or a digit.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the digit character class.

## Return values

**>0** The **mb* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismalnum() function" on page 4-54
"The ifx_gl_ismalpha() function" on page 4-55
"The ifx_gl_ismgraph() function"
"The ifx_gl_ismprint() function" on page 4-64
"The ifx_gl_ismxdigit() function" on page 4-70
"The ifx_gl_iswdigit() function" on page 4-76
"The ifx_gl_lc_errno() function" on page 4-85

# The ifx_gl_ismgraph() function

The **ifx_gl_ismgraph()** function determines whether a multibyte character contains a graphical (visible) character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismgraph(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismgraph()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismgraph()** function tests whether *mb* is in the graph character class according to the rules of the current locale. The graph character class contains all characters that have a visual representation, including characters from the alpha, digit, punct, and xdigit classes. In addition, all placeholder characters used in round-trip code-set conversion are in this class.

Characters in the graph class are also in the print class. No characters in the blank, space, or cntrl classes are in the graph class.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the graph character class.

### Return values

**>0** The **mb* character is in the graph character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismalnum() function" on page 4-54
"The ifx_gl_ismalpha() function" on page 4-55
"The ifx_gl_ismdigit() function" on page 4-59
"The ifx_gl_ismlower() function"
"The ifx_gl_ismprint() function" on page 4-64
"The ifx_gl_ismpunct() function" on page 4-65
"The ifx_gl_ismupper() function" on page 4-68
"The ifx_gl_ismxdigit() function" on page 4-70
"The ifx_gl_iswgraph() function" on page 4-77
"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_ismlower() function

The **ifx_gl_ismlower()** function determines whether a multibyte character contains a lowercase alphabetic character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismlower(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete

multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismlower()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismlower()** function tests whether *mb* is in the lower character class according to the rules of the current locale. The lower character class contains:

- the ASCII characters a through z.
- all other single-byte and multibyte lowercase characters for Latin-based languages; for example â, ç, é, and ü.

Characters in this class are also in the alpha, graph, and print classes. No characters in the upper, digit, blank, space, punct, or cntrl classes are in this class.

To determine the uppercase equivalent of an alphabetic multibyte character, you can use the **ifx_gl_tomupper()** function. Use the **ifx_gl_ismalpha()** function to test whether a multibyte character is an alphabetic character (uppercase or lowercase).

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the lower character class.

## Return values

**>0** The **mb* character is in the lower character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismalnum() function" on page 4-54
"The ifx_gl_ismalpha() function" on page 4-55
"The ifx_gl_ismgraph() function" on page 4-61
"The ifx_gl_ismprint() function"
"The ifx_gl_ismupper() function" on page 4-68
"The ifx_gl_iswlower() function" on page 4-78
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_tomupper() function" on page 4-116

# The ifx_gl_ismprint() function

The **ifx_gl_ismprint()** function determines whether a multibyte character contains a printable character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismprint(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismprint()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismprint()** function tests whether *mb* is in the print character class according to the rules of the current locale. The print character class contains all characters that have a visual representation or are in the space class but not in the cntrl class.

All characters from the alpha, digit, punct, xdigit, and graph classes are also in this class. No characters in the cntrl class are in this class.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the print character class.

### Return values

**>0** The **mb* character is in the print character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5

"Multibyte-character termination" on page 2-23

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismalnum() function" on page 4-54

"The ifx_gl_ismalpha() function" on page 4-55

"The ifx_gl_ismgraph() function" on page 4-61

"The ifx_gl_ismlower() function" on page 4-62

"The ifx_gl_ismprint() function" on page 4-64

"The ifx_gl_ismupper() function" on page 4-68

"The ifx_gl_ismxdigit() function" on page 4-70

"The ifx_gl_iswprint() function" on page 4-79

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_ismpunct() function

The **ifx_gl_ismpunct()** function determines whether a multibyte character contains a punctuation character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismpunct(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismpunct()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismpunct()** function tests whether *mb* is in the punct character class according to the rules of the current locale. The punct character class contains the following characters:

- The single-byte ASCII punctuation characters: ! @ # $ % ^ & * ( ) - _ = + \ | ‘ ~ [ ] { } ; : ' ” , . / < > ?
- Any non-ASCII punctuation characters that the locale might define

Graphic characters that are not really punctuation characters have traditionally been put in the punct class; instead they are in the graph class. Characters in this class are also in the graph and print classes. No characters in the upper, lower, alpha, digit, xdigit, blank, space, or cntrl classes are in this class.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the punct character class.

## Return values

**>0** The **mb* character is in the punct character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5

"Multibyte-character termination" on page 2-23

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismgraph() function" on page 4-61

"The ifx_gl_ismprint() function" on page 4-64

"The ifx_gl_iswpunct() function" on page 4-80

"The ifx_gl_lc_errno() function" on page 4-85

# The ifx_gl_ismspace() function

The **ifx_gl_ismspace()** function determines whether a multibyte character contains a space (vertical or horizontal) character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismspace(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismspace()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismspace()** function tests whether mb is in the space character class according to the rules of the current locale. The space character class contains:

- all characters from the blank class, including the horizontal space character (ASCII 0x020).
- the vertical-space character.
- the single-byte newline, vertical tab, form feed, and carriage return (ASCII 0x00A through 0x00D).
- any multibyte versions of newline, vertical tab, form feed, and carriage return.

No characters in the alpha, digit, xdigit, punct, or graph classes are in this class.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the space character class.

### Return values

**>0** The **mb* character is in the space character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5

"Multibyte-character termination" on page 2-23

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismblank() function" on page 4-57

"The ifx_gl_iswspace() function" on page 4-82

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_ismupper() function

The **ifx_gl_ismupper()** function determines whether a multibyte character contains an uppercase alphabetic character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismupper(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismupper()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_ismupper()** function tests whether mb is in the upper character class according to the rules of the current locale. The upper character class contains:

- the ASCII characters A through Z.
- all other single-byte and multibyte lowercase characters for Latin-based languages; for example: Â, Ç, É, and Ü.

Characters in this class are also in the alpha, graph, and print classes. No characters in the lower, digit, blank, space, punct, or cntrl classes are in this class.

To determine the lowercase equivalent of an alphabetic multibyte character, you can use the **ifx_gl_tomlower()** function. Use the **ifx_gl_ismalpha()** function to test whether a multibyte character is an alphabetic character (uppercase or lowercase).

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the upper character class.

## Return values

**>0** The *mb* character is in the upper character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismalnum() function" on page 4-54
"The ifx_gl_ismalpha() function" on page 4-55
"The ifx_gl_ismgraph() function" on page 4-61
"The ifx_gl_ismlower() function" on page 4-62
"The ifx_gl_ismprint() function" on page 4-64
"The ifx_gl_iswupper() function" on page 4-83
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_tomupper() function" on page 4-116

## The ifx_gl_ismxdigit() function

The **ifx_gl_ismxdigit()** function determines whether a multibyte character contains a hexadecimal digit.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_ismxdigit(mb, mb_byte_limit)
      gl_mchar_t *mb;
      int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_ismxdigit()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_ismxdigit()** function tests whether *mb* is in the xdigit character class according to the rules of the current locale. The xdigit character class contains:

- the 10 ASCII digit characters (ASCII 0x030 to 0x039).
- the characters `A` through `F`.
- the characters `a` through `f`.

Any multibyte versions or alternative representations of these hexadecimal digits that the locale might define (for example, Hindi or Kanji digits) are not in this class. Instead, they are in the alpha class.

Characters in this class are also in the graph and print classes. No characters in the blank, space, punct, or cntrl classes are in this class.

To determine whether a multibyte character contains a decimal digit, you can use the **ifx_gl_ismdigit()** function. Use the **ifx_gl_ismalnum()** function to test whether a multibyte character contains an alphabetic character or a digit.

### Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the xdigit character class.

### Return values

**>0** The *mb* character is in the xdigit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Character classification" on page 2-5

"Multibyte-character termination" on page 2-23

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_ismalnum() function" on page 4-54

"The ifx_gl_ismalpha() function" on page 4-55

"The ifx_gl_ismdigit() function" on page 4-59

"The ifx_gl_ismgraph() function" on page 4-61

"The ifx_gl_ismprint() function" on page 4-64

"The ifx_gl_iswxdigit() function" on page 4-84

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_iswalnum() function

The **ifx_gl_iswalnum()** function determines whether a wide character contains an alphabetic or digit character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswalnum(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_iswalnum()** function tests whether *wc* is in the alnum character class (either the alpha or digit class) according to the rules of the current locale. The alpha class contains all characters from the upper and lower classes. The digit class contains only the 10 ASCII digit characters: '0' to '9' (ASCII 0x030 to 0x039).

To determine whether a wide character is defined only in the alpha class, use the **ifx_gl_iswalpha()** function. To determine whether a wide character is defined only in the digit class, use the **ifx_gl_iswdigit()** function.

### Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the alnum character class.

### Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismalnum() function" on page 4-54
"The ifx_gl_iswalpha() function"
"The ifx_gl_iswdigit() function" on page 4-76
"The ifx_gl_iswlower() function" on page 4-78
"The ifx_gl_iswupper() function" on page 4-83
"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_iswalpha() function

The **ifx_gl_iswalpha()** function determines whether a wide character contains an alphabetic character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswalpha(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_iswalpha()** function tests whether *wc* is in the alpha character class according to the rules of the current locale. The alpha class contains all characters from the upper and lower classes. In addition to the uppercase and lowercase Latin-based characters, this class includes any alphabetic characters that the locale might define, including:

- Asian ideographic characters; for example, Kanji characters.
- Asian phonetic characters; for example, single-byte and multibyte Katakana and Hiragana characters.
- non-ASCII digit characters (see the digit class).
- Latin-based alphabetic characters that do not have a case-equivalent character.
- user-defined characters.
- vendor-defined characters.

Characters in the alpha class are also in the graph and print classes. No characters in the digit, blank, space, punct, or cntrl classes are in this class.

To determine the case of a wide alphabetic character, you can use the **ifx_gl_iswupper()** and **ifx_gl_iswlower()** functions. Use the **ifx_gl_iswalnum()** function to test whether a wide character is an alphabetic character or a digit.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the alpha character class.

## Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismalpha() function" on page 4-55
"The ifx_gl_iswalnum() function" on page 4-71
"The ifx_gl_iswgraph() function" on page 4-77
"The ifx_gl_iswlower() function" on page 4-78
"The ifx_gl_iswprint() function" on page 4-79
"The ifx_gl_iswupper() function" on page 4-83
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_tomlower() function" on page 4-114
"The ifx_gl_tomupper() function" on page 4-116

# The ifx_gl_iswblank() function

The **ifx_gl_iswblank()** function determines whether a wide character contains a horizontal-space character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswblank(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_iswblank()** function tests whether *wc* is in the blank character class according to the rules of the current locale. The blank character class includes the single-byte space (ASCII 0x020) and tab character (ASCII 0x009, ^I) plus any multibyte version of these characters that the locale defines. Characters in the blank class are also in the space class. No characters in the upper, lower, alpha, digit, xdigit, punct, or graph classes are in the blank class.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the blank character class.

## Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismblank() function" on page 4-57
"The ifx_gl_iswspace() function" on page 4-82
"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_iswcntrl() function

The **ifx_gl_iswcntrl()** function determines whether a wide character contains a control character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswcntrl(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_iswcntrl()** function tests whether *wc* is in the cntrl character class according to the rules of the current locale. The cntrl character class contains the single-byte control characters: alert, backspace, tab, newline, vertical tab, form feed, carriage return, NUL, SOH, STX, ETX, EOT, ENQ, ACK, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, IS4, IS3, IS2, IS1, and DEL. These characters are the ASCII characters whose code points are in the range 0x000 to 0x1F. In addition, the cntrl class contains any other control characters that the locale might define.

No characters in the upper, lower, alpha, digit, xdigit, punct, graph, or print classes are in the cntrl class.

### Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the cntrl character class.

### Return values

**>0** The *wc* character is in the cntrl character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismcntrl() function" on page 4-58
"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_iswdigit() function

The **ifx_gl_iswdigit()** function determines whether a wide character contains a decimal digit.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswdigit(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_iswdigit()** function tests whether *wc* is in the digit character class according to the rules of the current locale. The digit character class contains only the 10 ASCII digit characters (ASCII 0x030 to 0x039). Any multibyte versions or alternative representations of these digits that the locale might define (for example, Hindi or Kanji digits) are not in this class. Instead, they are in the alpha class.

Characters in this class are also in the xdigit, graph, and print classes. No characters in the upper, lower, alpha, blank, space, punct, or cntrl classes are in this class.

To determine whether a wide character contains a hexadecimal digit, you can use the **ifx_gl_iswxdigit()** function. Use the **ifx_gl_iswalnum()** function to test whether a wide character is an alphabetic character or a digit.

### Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the digit character class.

### Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismdigit() function" on page 4-59
"The ifx_gl_iswalnum() function" on page 4-71
"The ifx_gl_iswalpha() function" on page 4-72
"The ifx_gl_iswgraph() function"
"The ifx_gl_iswprint() function" on page 4-79
"The ifx_gl_iswxdigit() function" on page 4-84
"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_iswgraph() function

The **ifx_gl_iswgraph()** function determines whether a wide character contains a graphical (visible) character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswgraph(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_iswgraph()** function tests whether *wc* is in the graph character class according to the rules of the current locale. The graph character class contains all characters that have a visual representation, including characters from the alpha, digit, punct, and xdigit classes. In addition, all placeholder characters used in round-trip code-set conversion are in this class.

Characters in the graph class are also in the print class. No characters in the blank, space, or cntrl classes are in the graph class.

### Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the graph character class.

### Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismgraph() function" on page 4-61
"The ifx_gl_iswalnum() function" on page 4-71
"The ifx_gl_iswalpha() function" on page 4-72
"The ifx_gl_iswdigit() function" on page 4-76
"The ifx_gl_iswprint() function" on page 4-79
"The ifx_gl_iswpunct() function" on page 4-80
"The ifx_gl_iswupper() function" on page 4-83
"The ifx_gl_iswxdigit() function" on page 4-84
"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_iswlower() function

The **ifx_gl_iswlower()** function determines whether a wide character contains a lowercase alphabetic character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswlower(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_iswlower()** function tests whether *wc* is in the lower character class according to the rules of the current locale. The lower character class contains:

- the ASCII characters a through z.

- all other single-byte and multibyte lowercase characters for Latin-based languages; for example: â, ç, é, and ü.

Characters in this class are also in the alpha, graph, and print classes. No characters in the upper, digit, blank, space, punct, or cntrl classes are in this class.

To obtain the uppercase equivalent of an alphabetic wide character, you can use the **ifx_gl_towupper()** function. Use the **ifx_gl_iswalpha()** function to test whether a wide character is an alphabetic character (uppercase or lowercase).

### Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the lower character class.

### Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismlower() function" on page 4-62
"The ifx_gl_iswalnum() function" on page 4-71
"The ifx_gl_iswalpha() function" on page 4-72
"The ifx_gl_iswgraph() function" on page 4-77
"The ifx_gl_iswprint() function"
"The ifx_gl_iswupper() function" on page 4-83
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_towupper() function" on page 4-118

## The ifx_gl_iswprint() function

The **ifx_gl_iswprint()** function determines whether a wide character contains a printable character.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswprint(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_iswprint()** function tests whether *wc* is in the print character class according to the rules of the current locale. The print character class contains all characters that have a visual representation or are in the space class but not in the cntrl class.

All characters from the alpha, digit, punct, xdigit, and graph classes are also in this class. No characters in the cntrl class are in this class.

### Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the print character class.

### Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismprint() function" on page 4-64
"The ifx_gl_iswalnum() function" on page 4-71
"The ifx_gl_iswalpha() function" on page 4-72
"The ifx_gl_iswdigit() function" on page 4-76
"The ifx_gl_iswgraph() function" on page 4-77
"The ifx_gl_iswlower() function" on page 4-78
"The ifx_gl_iswpunct() function"
"The ifx_gl_iswupper() function" on page 4-83
"The ifx_gl_iswxdigit() function" on page 4-84
"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_iswpunct() function

The **ifx_gl_iswpunct()** function determines whether a wide character contains a punctuation character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswpunct(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_iswpunct()** function tests whether *wc* is in the punct character class according to the rules of the current locale. The punct character class contains the following characters:

- The single-byte ASCII punctuation characters: ! @ # $ % ^ & * ( ) - _ = + \ | ‘ ~ [ ] { } ; : ' ” , . / < > ?
- Any non-ASCII punctuation characters that the locale might define

Graphic characters that are not true punctuation characters have traditionally been put in the punct class; instead they are in the graph class. Characters in this class are also in the graph and print classes. No characters in the upper, lower, alpha, digit, xdigit, blank, space, or cntrl classes are in this class.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the punct character class.

## Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismpunct() function" on page 4-65

"The ifx_gl_iswgraph() function" on page 4-77

"The ifx_gl_iswprint() function" on page 4-79

"The ifx_gl_lc_errno() function" on page 4-85

# The ifx_gl_iswspace() function

The **ifx_gl_iswspace()** function determines whether a wide character contains a space (vertical or horizontal) character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswspace(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_iswspace()** function tests whether *wc* is in the space character class according to the rules of the current locale. The space character class contains:

- all characters from the blank class, including the horizontal space character (ASCII 0x020).
- the vertical-space character.
- the single-byte newline, vertical tab, form feed, and carriage return (ASCII 0x0A through 0x0D).
- any multibyte versions of newline, vertical tab, form feed, and carriage return.

No characters in the alpha, digit, xdigit, punct, or graph classes are in this class.

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the space character class.

## Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismspace() function" on page 4-67
"The ifx_gl_iswblank() function" on page 4-74
"The ifx_gl_lc_errno() function" on page 4-85

# The ifx_gl_iswupper() function

The **ifx_gl_iswupper()** function determines whether a wide character contains an uppercase alphabetic character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswupper(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_iswupper()** function tests whether *wc* is in the upper character class according to the rules of the current locale. The upper character class contains:

- the ASCII characters A through Z.
- all other single-byte and multibyte uppercase characters for Latin-based languages; for example: Â, Ç, É, and Ü.

Characters in this class are also in the alpha, graph, and print classes. No characters in the lower, digit, blank, space, punct, or cntrl classes are in this class.

To determine the lowercase equivalent of an alphabetic wide character, you can use the **ifx_gl_tomlower()** function. Use the **ifx_gl_ismalpha()** function to test whether a wide character is an alphabetic character (uppercase or lowercase).

## Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the upper character class.

## Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismupper() function" on page 4-68
"The ifx_gl_iswalnum() function" on page 4-71
"The ifx_gl_iswalpha() function" on page 4-72
"The ifx_gl_iswgraph() function" on page 4-77
"The ifx_gl_iswlower() function" on page 4-78
"The ifx_gl_iswprint() function" on page 4-79
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_towupper() function" on page 4-118

## The ifx_gl_iswxdigit() function

The **ifx_gl_iswxdigit()** function determines whether a wide character contains a hexadecimal digit.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_iswxdigit(wc)
       gl_wchar_t *wc;
```

*wc* A pointer to the wide character whose character classification you want to determine.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_iswxdigit()** function tests whether *wc* is in the xdigit character class according to the rules of the current locale. The xdigit character class contains:

- the 10 ASCII digit characters (ASCII 0x030 to 0x039).
- the characters `A` through `F`.
- the characters `a` through `f`.

Any multibyte versions or alternative representations of these hexadecimal digits that the locale might define (for example, Hindi or Kanji digits) are not in this class. Instead, they are in the alpha class.

Characters in this class are also in the graph and print classes. No characters in the blank, space, punct, or cntrl classes are in this class.

To determine whether a wide character contains a decimal digit, you can use the **ifx_gl_iswdigit()** function. Use the **ifx_gl_iswalnum()** function to test whether a wide character contains an alphabetic character or a digit.

### Locale information

The LC_CTYPE category of the current locale affects the behavior of this function because it defines the xdigit character class.

### Return values

**>0** The *wc* character is in the digit character class.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"The ifx_gl_ismxdigit() function" on page 4-70
"The ifx_gl_iswalnum() function" on page 4-71
"The ifx_gl_iswalpha() function" on page 4-72
"The ifx_gl_iswdigit() function" on page 4-76
"The ifx_gl_iswgraph() function" on page 4-77
"The ifx_gl_iswprint() function" on page 4-79
"The ifx_gl_lc_errno() function"

## The ifx_gl_lc_errno() function

The **ifx_gl_lc_errno()** function returns the value of the error number that some other IBM Informix GLS function has set.

### Syntax

```
#include <gls.h>
...
int ifx_gl_lc_errno(void);
```

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The Informix GLS functions set the *error number*, of type **int**, to indicate additional information about an error that occurred. Informix GLS functions set this value only when an error occurs (unless documented otherwise). Because any Informix

GLS function can set the error number, you must use **ifx_gl_lc_errno()** to inspect the error number immediately after you call the function in which the error occurred.

**Tip:** The Errors section for each function documents the values that **ifx_gl_lc_errno()** returns after that particular function.

Certain code-set conversion functions are not dependent on a locale and do not set an error number that you can retrieve with **ifx_gl_lc_errno()**. Instead, these functions return a negative error code.

### Return values

**Integer error number**
The error number that an Informix GLS function has set. These error numbers are defined in the `gls.h` file and are described in the Errors section for each function.

**-1** The function was not successful. The current processing locale has not been correctly initialized.

### Errors

None

**Related concepts**:

"Informix GLS exceptions" on page 1-8

## The ifx_gl_mb_loc_max() function

The **ifx_gl_mb_loc_max()** function returns the maximum number of bytes in the characters of the current locale.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mb_loc_max(void);;
```

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_mb_loc_max()** function is useful when you need to allocate memory for a multibyte-character string. It returns the maximum number of bytes that any multibyte character can occupy. A multibyte code set can have characters of one, two, three, or four bytes.

### Return values

**>=0** The maximum number of bytes in the current locale.

**-1** The function was not successful.

### Errors

None

**Related reference**:

"Managing memory for strings and characters" on page 2-24

# The ifx_gl_mblen() function

The **ifx_gl_mblen()** function returns the number of bytes in a multibyte character.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mblen(mb, mb_byte_limit)
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*mb* A pointer to the multibyte character whose character classification you want to determine.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mblen()** function determines the number of bytes in the multibyte character, *mb*. This function is often used to advance a pointer through a null-terminated multibyte string. However, it is slightly faster to advance a pointer through a null-terminated string with **ifx_gl_mbsnext()**, as the following code shows:

```
for ( mb = mbs; *mb != '\0'; )
    {
    if ( (mb = ifx_gl_mbsnext(mb, IFX_GL_NO_LIMIT)) == NULL )
        /* handle error */
    }
```

## Return values

**>=0** The number of bytes in the mb multibyte character.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"String traversal" on page 2-18
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbsnext() function" on page 4-101
"The ifx_gl_mbsprev() function" on page 4-106

# The ifx_gl_mbscat() function

The **ifx_gl_mbscat()** function concatenates two multibyte-character strings.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbscat(mbs1, mbs1_byte_length, mbs2, mbs2_byte_length)
       gl_mchar_t *mbs1;
       int mbs1_byte_length;
       gl_mchar_t *mbs2;
       int mbs2_byte_length;
```

*mbs1* A pointer to the multibyte-character string to which the function concatenates *mbs2*.

*mbs1_byte_length*
The integer number of bytes in the *mbs1* string. If *mbs1_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs1* is a null-terminated string.

*mbs2* A pointer to the multibyte-character string to concatenate onto *mbs1*.

*mbs2_byte_length*
The integer number of bytes in the *mbs2* string. If *mbs2_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbscat()** function appends a copy of the *mbs2* multibyte-character string to the end of the *mbs1* multibyte-character string.

If *mbs1* and *mbs2* overlap, the results of this function are undefined. If *mbs1_byte_length* is equal to IFX_GL_NULL, the function null-terminates the concatenated string. Any other value of *mbs1_byte_length* means that the function does not null-terminate the resulting concatenated string.

## Return values

**>=0** The number of characters in the concatenated string, not including any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *mbs1_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *mbs2_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *mbs1_byte_length* is equal to IFX_GL_NULL, and *mbs2_byte_length* is greater than or equal to 0; or *mbs1_byte_length* is greater than or equal to 0, and *mbs2_byte_length* is equal to IFX_GL_NULL.

**IFX_GL_EILSEQ**
Either *mbs1* or *mbs2* contains an invalid multibyte character.

**IFX_GL_EINVAL**
Either the function cannot determine whether the last character of *mbs1* is a valid multibyte character because it would need to read more than *mbs1_byte_length* bytes from *mbs1*, or the function cannot determine whether the last character of *mbs2* is a valid multibyte character because it would need to read more than *mbs2_byte_length* bytes from *mbs2*.

**Related concepts**:

"Concatenation" on page 2-19

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbsncat() function" on page 4-98

"The ifx_gl_wcscat() function" on page 4-119

"The ifx_gl_wcsncat() function" on page 4-126

## The ifx_gl_mbschr() function

The **ifx_gl_mbschr()** function searches for the first occurrence of a character in a multibyte-character string.

### Syntax

```
#include <ifxgls.h>
...
gl_mchar_t *ifx_gl_mbschr(mbs, mbs_byte_length, mb, mb_byte_limit)
        gl_mchar_t *mbs;
        int mbs_byte_length;
        gl_mchar_t *mb;
        int mb_byte_limit;
```

*mbs* A pointer to the multibyte-character string in which the function searches for *mb*.

*mbs_byte_length*
The integer number of bytes in the *mbs* string. If *mbs_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs1* is a null-terminated string.

*mb* A pointer to the multibyte character to search for in *mbs*.

*mb_byte_limit*
The number of bytes to read from the *mb* character to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the **ifx_gl_mbschr()** function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbschr()** function locates the first occurrence of *mb* in the multibyte string *mbs*.

## Return values

**gl_mchar_t ***
A pointer to the first occurrence of *mb* in *mbs*.

**NULL** Either *mb* was not found in *mbs* (the **ifx_gl_lc_errno()** error number is set to 0); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns NULL and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is an invalid multibyte character, or **mbs* contains an invalid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*, or the function cannot determine whether the last character of *mbs* is a valid multibyte character because it would need to read more than *mbs_byte_length* bytes from *mbs*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**IFX_GL_PARAMERR**
The *mbs_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**Related concepts**:

"Character-string termination" on page 2-22
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"Character searching" on page 2-20
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbsrchr() function" on page 4-107
"The ifx_gl_wcschr() function" on page 4-120
"The ifx_gl_wcsrchr() function" on page 4-131

# The ifx_gl_mbscoll() function

The **ifx_gl_mbscoll()** function uses locale-specific order to compare two multibyte-character strings.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbscoll(mbs1, mbs1_byte_length, mbs2, mbs2_byte_length)
       gl_mchar_t *mbs1;
       int mbs1_byte_length;
       gl_mchar_t *mbs2;
       int mbs2_byte_length;
```

*mbs1* A pointer to the multibyte-character string to compare with *mbs2*.

*mbs1_byte_length*
The integer number of bytes in the *mbs1* string. If *mbs1_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs1* is a null-terminated string.

*mbs2* A pointer to the multibyte-character string to compare with *mbs1*.

*mbs2_byte_length*
The integer number of bytes in the *mbs2* string. If *mbs2_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbscoll()** function uses locale-specific order to compare the multibyte-character strings *mbs1* and *mbs2*.

## Locale information

The LC_COLLATE category of the current locale affects the behavior of this function because it defines the locale-specific order.

## Return values

**<0** The *mbs1* string is less than the *mbs2* string (*mbs1* < *mbs2*); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

**=0** The *mbs1* string is equal to the *mbs2* string (*mbs1 = mbs2*).

**>0** The *mbs1* string is greater than the *mbs2* string (*mbs1 > mbs2*).

### Errors

This function does not return a unique value to indicate an error. If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *mbs1_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *mbs2_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *mbs1_byte_length* is equal to IFX_GL_NULL, and *mbs2_byte_length* is greater than or equal to 0; or *mbs1_byte_length* is greater than or equal to 0, and *mbs2_byte_length* is equal to IFX_GL_NULL.

**IFX_GL_EILSEQ**
Either *mbs1* or *mbs2* contains an invalid multibyte character.

**IFX_GL_EINVAL**
Either the function cannot determine whether the last character of *mbs1* is a valid multibyte character because it would need to read more than *mbs1_byte_length* bytes from *mbs1*, or the function cannot determine whether the last character of *mbs2* is a valid multibyte character because it would need to read more than *mbs2_byte_length* bytes from *mbs2*.

**IFX_GL_ENOMEM**
Not enough memory is available to complete the operation.

**Related concepts**:

"Character/string comparison and sorting" on page 2-20

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_wcscoll() function" on page 4-121

## The ifx_gl_mbscpy() function

The **ifx_gl_mbscpy()** function copies a multibyte-character string.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbscpy(mbs1, mbs2, mbs2_byte_length)
       gl_mchar_t *mbs1;
       gl_mchar_t *mbs2;
       int mbs2_byte_length;
```

*mbs1* A pointer to the location where the function copies *mbs2*.

*mbs2* A pointer to the multibyte-character string to copy to *mbs1*.

*mbs2_byte_length*
The integer number of bytes in the *mbs2* string. If *mbs2_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs2* is a null-terminated

string. For more general information about string lengths, see "Character-string termination" on page 2-22.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_mbscpy()** function copies the multibyte string *mbs2* to the location that *mbs1* references.

If *mbs1* and *mbs2* overlap, the results of this function are undefined. If *mbs2_byte_length* is equal to IFX_GL_NULL, the function null-terminates the concatenated string. Any other value of *mbs2_byte_length* means that the function does not null-terminate the resulting concatenated string.

### Return values

**>=0** The number of bytes in the copied string, not including any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
The *mbs2_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_EILSEQ**
The **mbs2* value contains an invalid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether the last character of *mbs2* is a valid multibyte character because it would need to read more than *mbs2_byte_length* bytes from *mbs2*.

**Related concepts**:

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"String copying" on page 2-19
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbsncpy() function" on page 4-99
"The ifx_gl_wcscpy() function" on page 4-123
"The ifx_gl_wcsncpy() function" on page 4-128

## The ifx_gl_mbscspn() function

The **ifx_gl_mbscspn()** function determines the length of a complementary multibyte substring for a multibyte-character string.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbscspn(mbs1, mbs1_byte_length, mbs2, mbs2_byte_length)
       gl_mchar_t *mbs1;
       int mbs1_byte_length;
       gl_mchar_t *mbs2;
       int mbs2_byte_length;
```

*mbs1* A pointer to the multibyte-character string to check for the complementary string of characters in *mbs2*.

*mbs1_byte_length*
The integer number of bytes in the *mbs1* string. If *mbs1_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs1* is a null-terminated string.

*mbs2* A pointer to the string of multibyte characters to search for in *mbs1*.

*mbs2_byte_length*
The integer number of bytes in the *mbs2* string. If *mbs2_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbscspn()** function returns the size of the initial substring of *mbs1* that does not contain any characters that match characters in *mbs2*. The initial substring begins at the first character of *mbs1*. Therefore, this size is the number of characters in *mbs1* before the first character that the *mbs1* and *mbs2* strings have in common. The string of unmatched characters is called the *complementary string*.

For example, suppose you have the following two multibyte-characters strings, `mbs1` and `mbs2`:

```
mbs1 = "A1A2B1B2B3C1C2D1D2D3A1A2E1F1F2A1A2G1D1D2D3";
mbs2 = "F1F2G1D1D2D3";
```

With these multibyte strings, the following call to the **ifx_gl_mbscspn()** function returns 3:

```
ifx_gl_mbscspn(mbs1, bytelen1, mbs2, bytelen2)
```

The first three characters of `mbs1` are not in `mbs2`. The fourth character in `mbs1` is $D^1D^2D^3$, which is in `mbs2` also.

## Return values

**>=0** The number of characters in the initial substring of *mbs1* that consists entirely of multibyte characters not in the string *mbs2*.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *mbs1_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *mbs2_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *mbs1_byte_length* is equal to IFX_GL_NULL, and *mbs2_byte_length* is greater than or equal to 0; or *mbs1_byte_length* is greater than or equal to 0, and *mbs2_byte_length* is equal to IFX_GL_NULL.

**IFX_GL_EILSEQ**
Either *mbs1* or *mbs2* contains an invalid multibyte character.

**IFX_GL_EINVAL**
Either the function cannot determine whether the last character of *mbs1* is a valid multibyte character because it would need to read more than *mbs1_byte_length* bytes from *mbs1*, or the function cannot determine whether the last character of *mbs2* is a valid multibyte character because it would need to read more than *mbs2_byte_length* bytes from *mbs2*.

**Related concepts**:

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"String-length determination" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbsspn() function" on page 4-109

"The ifx_gl_wcscspn() function" on page 4-124

"The ifx_gl_wcsspn() function" on page 4-132

# The ifx_gl_mbslen() function

The **ifx_gl_mbslen()** function determines the number of characters in a multibyte-character string.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbslen(mbs, mbs_byte_length)
       gl_mchar_t *mbs;
       int mbs_byte_length;
```

*mbs* A pointer to the multibyte-character string whose length the function determines.

*mbs_byte_length*
The integer number of bytes in the *mbs* string. If *mbs_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbslen()** function returns the number of characters (not bytes) in the string *mbs*, not including any terminating null character. The length that

**ifx_gl_mbslen()** returns includes any trailing white space. The trailing-space characters are all characters that the blank character class of the current locale defines. Use the ifx_gl_mbntslen() function if you want the length without trailing white space.

For example, the following call to **ifx_gl_mbslen()** returns a value of 12 (where $A^1A^2$ is the multibyte character of 2 bytes, $B^1B^2B^3$ is the multibyte character of 3 bytes, and $E^1$ is the multibyte character of 1 byte, and each *s* represents a single-byte horizontal white space character):

```
ifx_gl_mbslen("A1A2B1B2B3scdsE1sssss", mbs_byte_length)
```

Use the **ifx_gl_mbsntslen()** or **ifx_gl_mbsntsbytes()** function if you want the length without trailing white space.

### Return values

**>=0** The number of characters in the *mbs* string, not including any null terminator but including any trailing white space.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
The *mbs_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_EILSEQ**
The **mbs* value contains an invalid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether the last character of *mbs* is a valid multibyte character because it would need to read more than *mbs_byte_length* bytes from *mbs*.

**Related concepts**:

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"String-length determination" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mblen() function" on page 4-87

"The ifx_gl_mbsnext() function" on page 4-101

"The ifx_gl_mbsntsbytes() function" on page 4-102

"The ifx_gl_mbsntslen() function" on page 4-103

"The ifx_gl_wcslen() function" on page 4-125

## The ifx_gl_mbsmbs() function

The **ifx_gl_mbsmbs()** function searches for a specified substring within a multibyte-character string.

## Syntax

```
#include <ifxgls.h>
...
gl_mchar_t *ifx_gl_mbsmbs(mbs1, mbs1_byte_length, mbs2, mbs2_byte_length)
        gl_mchar_t *mbs1;
        int mbs1_byte_length;
        gl_mchar_t *mbs2;
        int mbs2_byte_length;
```

*mbs1* A pointer to the multibyte-character string to search for the substring *mbs2*.

*mbs1_byte_length*
The integer number of bytes in the *mbs1* string. If *mbs1_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs1* is a null-terminated string.

*mbs2* A pointer to the string of multibyte characters to search for in *mbs1*.

*mbs2_byte_length*
The integer number of bytes in the *mbs2* string. If *mbs2_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbsmbs()** function searches for the first occurrence of the multibyte string *mbs2* in the multibyte string *mbs1*.

## Return values

**gl_mchar_t ***
A pointer to the first byte of the first occurrence of *mbs2* in *mbs1*.

**NULL** Either *mbs2* was not found in *mbs1* (the **ifx_gl_lc_errno()** error number is set to 0); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns NULL and sets **ifx_gl_lc_errno()** to one of the following values.

**IFX_GL_PARAMERR**
Either *mbs1_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *mbs2_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *mbs1_byte_length* is equal to IFX_GL_NULL, and *mbs2_byte_length* is greater than or equal to 0; or *mbs1_byte_length* is greater than or equal to 0, and *mbs2_byte_length* is equal to IFX_GL_NULL.

**IFX_GL_EILSEQ**
Either *mbs1* or *mbs2* contains an invalid multibyte character.

**IFX_GL_EINVAL**
Either the function cannot determine whether the last character of *mbs1* is a valid multibyte character because it would need to read more than

*mbs1_byte_length* bytes from *mbs1*, or the function cannot determine whether the last character of *mbs2* is a valid multibyte character because it would need to read more than *mbs2_byte_length* bytes from *mbs2*.

**Related concepts**:

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"Character searching" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbschr() function" on page 4-89

"The ifx_gl_mbsrchr() function" on page 4-107

"The ifx_gl_wcschr() function" on page 4-120

"The ifx_gl_wcswcs() function" on page 4-135

# The ifx_gl_mbsncat() function

The **ifx_gl_mbsncat()** function concatenates a specified number of multibyte characters in one multibyte string to another.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbsncat(mbs1, mbs1_byte_length, mbs2, mbs2_byte_length, char_limit)
       gl_mchar_t *mbs1;
       int mbs1_byte_length;
       gl_mchar_t *mbs2;
       int mbs2_byte_length;
       int char_limit;
```

*mbs1* A pointer to the multibyte-character string to which the function concatenates *mbs2*.

*mbs1_byte_length*
The integer number of bytes in the *mbs1* string. If *mbs1_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs1* is a null-terminated string.

*mbs2* A pointer to the multibyte-character string to concatenate onto *mbs1*.

*mbs2_byte_length*
The integer number of bytes in the *mbs2* string. If *mbs2_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs2* is a null-terminated string.

*char_limit*
The maximum number of multibyte characters to read from *mbs2*.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbscat()** function appends *mbs2* to the end of *mbs1*.

If *mbs1* and *mbs2* overlap, the results of this function are undefined. If *mbs1_byte_length* is equal to IFX_GL_NULL, the function null-terminates the

concatenated string. Any other value of *mbs1_byte_length* means that the function does not null-terminate the resulting concatenated string.

### Return values

**>=0** The number of characters in the concatenated string, not including any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *mbs1_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *mbs2_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *mbs1_byte_length* is equal to IFX_GL_NULL, and *mbs2_byte_length* is greater than or equal to 0; or *mbs1_byte_length* is greater than or equal to 0, and *mbs2_byte_length* is equal to IFX_GL_NULL.

**IFX_GL_EILSEQ**
Either *mbs1* or *mbs2* contains an invalid multibyte character.

**IFX_GL_EINVAL**
Either the function cannot determine whether the last character of *mbs1* is a valid multibyte character because it would need to read more than *mbs1_byte_length* bytes from *mbs1*, or the function cannot determine whether the last character of *mbs2* is a valid multibyte character because it would need to read more than *mbs2_byte_length* bytes from *mbs2*.

**Related concepts**:

"Concatenation" on page 2-19

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbscat() function" on page 4-88

"The ifx_gl_wcscat() function" on page 4-119

"The ifx_gl_wcsncat() function" on page 4-126

## The ifx_gl_mbsncpy() function

The **ifx_gl_mbsncpy()** function copies a specified number of multibyte characters from a multibyte-character string.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbsncpy(mbs1, mbs2, mbs2_byte_length, char_limit)
       gl_mchar_t *mbs1;
       gl_mchar_t *mbs2;
       int mbs2_byte_length;
       int char_limit;
```

*mbs1* A pointer to the location where the function copies *mbs2*.

*mbs2* A pointer to the multibyte-character string to copy to *mbs1*.

*mbs2_byte_length*
The integer number of bytes in the *mbs2* string. If *mbs2_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs2* is a null-terminated string.

*char_limit*
The maximum number of multibyte characters to read from *mbs2*.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbsncpy()** function copies *mbs2* to the location that *mbs1* references. The function reads no more than *char_limit* characters from *mbs2* and writes them to *mbs1*. If *mbs1* and *mbs2* overlap, the results of this function are undefined.

If *mbs1* and *mbs2* overlap, the results of this function are undefined. If *mbs2_byte_length* is equal to IFX_GL_NULL, the function null-terminates the concatenated string. Any other value of *mbs2_byte_length* means that the function does not null-terminate the resulting concatenated string.

## Return values

**>=0** The number of bytes in the copied string, not including any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
The *mbs2_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_EILSEQ**
The **mbs2* value contains an invalid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether the last character of *mbs2* is a valid multibyte character because it would need to read more than *mbs2_byte_length* bytes from *mbs2*.

**Related concepts**:

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"String copying" on page 2-19

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbscpy() function" on page 4-92

"The ifx_gl_wcscpy() function" on page 4-123

"The ifx_gl_wcsncpy() function" on page 4-128

# The ifx_gl_mbsnext() function

The **ifx_gl_mbsnext()** function returns the next multibyte character from a multibyte string.

## Syntax

```
#include <ifxgls.h>
...
gl_mchar_t *ifx_gl_mbsnext(mb, mb_byte_limit)
        gl_mchar_t *mb;
        int mb_byte_limit;
```

*mb* A pointer to the current multibyte character in the multibyte string.

*mb_byte_limit*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbsnext()** function returns a pointer to the next multibyte character after mb in a multibyte-character string. This function is typically used to transform the single-byte forward-traversal loop to a multibyte loop. For example, suppose you have the following single-byte loop:

```
for ( sb = sbs; *sb != '\0'; sb++ )
    {
    /* Process single-byte character */
```

The following code fragment traverses the string with the **ifx_gl_mbsnext()** function instead of incrementing the pointer of the single-byte loop:

```
for ( mb = mbs; *mb != '\0'; )
    {
    /* Process multibyte character; increment pointer through string */
    if ( (mb = ifx_gl_mbsnext(mb, IFX_GL_NO_LIMIT)) == NULL )
    /* Handle error */
    }
```

## Return values

**gl_mchar_t ***
A pointer to the byte that immediately follows *mb*.

**NULL** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns NULL and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"String traversal" on page 2-18

"Multibyte-character termination" on page 2-23

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mblen() function" on page 4-87

"The ifx_gl_mbsprev() function" on page 4-106

## The ifx_gl_mbsntsbytes() function

The **ifx_gl_mbsntsbytes()** function determines the number of bytes in a multibyte string. It ignores trailing spaces.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbsntsbytes(mbs, mbs_byte_length)
       gl_mchar_t *mbs;
       int mbs_byte_length;
```

*mbs* A pointer to the multibyte-character string whose length the function determines.

*mbs_byte_length*
The integer number of bytes in the *mbs* string. If *mbs_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_mbsntsbytes()** function returns the number of bytes in *mbs*, not including the trailing-space characters. The trailing-space characters are all characters that the blank character class of the current locale defines. For more information about the blank class, see "Character classification" on page 2-5.

Space characters that are embedded in the string at any place except the end of the string are included in the count. For example, the following call to **ifx_gl_mbsntsbytes()** returns a value of 10 (the number of bytes in the string "$A^1A^2B^1B^2B^3$ cd $E^1$", where $A^1A^2$ is the multibyte character of 2 bytes, $B^1B^2B^3$ is the multibyte character of 3 bytes, and $E^1$ is the multibyte character of 1 byte, and *s* represents a single-byte horizontal white space character):

```
ifx_gl_mbsntsbytes("A¹A²B¹B²B³scdsE¹ssssss", mbs_byte_length)
```

### Return values

**>=0** The number of bytes in the *mbs* string, not including any trailing space.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
The *mbs_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_EILSEQ**
The **mbs* value contains an invalid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether the last character of *mbs* is a valid multibyte character because it would need to read more than *mbs_byte_length* bytes from *mbs*.

**Related concepts**:

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"String-length determination" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbsntslen() function"

"The ifx_gl_wcslen() function" on page 4-125

## The ifx_gl_mbsntslen() function

The **ifx_gl_mbsntslen()** function determines the number of characters in a multibyte string. It ignores trailing spaces.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbsntslen(mbs, mbs_byte_length)
       gl_mchar_t *mbs;
       int mbs_byte_length;
```

*mbs* A pointer to the multibyte-character string whose length the function determines.

*mbs_byte_length*
The integer number of bytes in the *mbs* string. If *mbs_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbsntslen()** function returns the number of characters in *mbs*, not including any trailing-space characters. The trailing-space characters are all characters that the blank character class of the current locale defines. For more information about the blank class, see "Character classification" on page 2-5.

Space characters that are embedded in the string at any place except the end of the string are included in the count. For example, the following call to **ifx_gl_mbsntslen()** returns a value of 7 (the number of bytes in the string "$A^1A^2B^1B^2B^3$ cd $E^1$", where $A^1A^2$ is the multibyte character of 2 bytes, $B^1B^2B^3$ is the multibyte character of 3 bytes, and $E^1$ is the multibyte character of 1 byte, and *s* represents a single-byte horizontal white space character):

```
ifx_gl_mbsntslen("A¹A²B¹B²B³scdsE¹ssssss", mbs_byte_length)
```

Use the **ifx_gl_mbslen()** function if you want the length to include trailing white space.

## Return values

**>=0** The number of bytes in the *mbs* string, not including any trailing space.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
The *mbs_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_EILSEQ**
The **mbs* value contains an invalid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether the last character of *mbs* is a valid multibyte character because it would need to read more than *mbs_byte_length* bytes from *mbs*.

**Related concepts**:

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"String-length determination" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mblen() function" on page 4-87

"The ifx_gl_mbsntsbytes() function" on page 4-102

"The ifx_gl_wcsntslen() function" on page 4-129

# The ifx_gl_mbspbrk() function

The **ifx_gl_mbspbrk()** function searches for any multibyte character from one multibyte string in another multibyte-character string.

## Syntax

```
#include <ifxgls.h>
...
gl_mchar_t *ifx_gl_mbspbrk(mbs1, mbs1_byte_length, mbs2, mbs2_byte_length)
        gl_mchar_t *mbs1;
        int mbs1_byte_length;
        gl_mchar_t *mbs2;
        int mbs2_byte_length;
```

*mbs1* A pointer to the string of multibyte characters to search for the characters in *mbs2*.

*mbs1_byte_length*
The integer number of bytes in the *mbs1* string. If *mbs1_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs1* is a null-terminated string.

*mbs2* A pointer to the multibyte-character string whose characters the function searches for in *mbs1*.

*mbs2_byte_length*
The integer number of bytes in the *mbs2* string. If *mbs2_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbspbrk()** function searches for the first occurrence of any multibyte character from the string *mbs2* in the multibyte string *mbs1*.

## Return values

**gl_mchar_t ***
A pointer to the first byte of the first occurrence in *mbs1* of any character from *mbs2*.

**NULL** Either no character in *mbs2* was found in *mbs1* (the **ifx_gl_lc_errno()** error number is set to 0); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns NULL and sets **ifx_gl_lc_errno()** to one of the following values.

**IFX_GL_PARAMERR**
Either *mbs1_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *mbs2_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *mbs1_byte_length* is equal to IFX_GL_NULL, and *mbs2_byte_length* is greater than or equal to 0; or *mbs1_byte_length* is greater than or equal to 0, and *mbs2_byte_length* is equal to IFX_GL_NULL.

**IFX_GL_EILSEQ**
Either *mbs1* or *mbs2* contains an invalid multibyte character.

**IFX_GL_EINVAL**
Either the function cannot determine whether the last character of *mbs1* is a valid multibyte character because it would need to read more than *mbs1_byte_length* bytes from *mbs1*, or the function cannot determine whether the last character of *mbs2* is a valid multibyte character because it would need to read more than *mbs2_byte_length* bytes from *mbs2*.

**Related concepts**:

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"Character searching" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbschr() function" on page 4-89

"The ifx_gl_mbsrchr() function" on page 4-107

"The ifx_gl_wcschr() function" on page 4-120

"The ifx_gl_wcspbrk() function" on page 4-130

"The ifx_gl_wcsrchr() function" on page 4-131

## The ifx_gl_mbsprev() function

The **ifx_gl_mbsprev()** function returns the previous multibyte character in a multibyte string.

### Syntax

```
#include <ifxgls.h>
...
gl_mchar_t *ifx_gl_mbsprev(mbs_start, mb)
        gl_mchar_t *mbs_start;
        gl_mchar_t *mb;
```

*mbs_start*
A pointer to beginning of the multibyte string.

*mb* A pointer to the current multibyte character in the multibyte string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_mbsprev()** function returns a pointer to the multibyte character before *mb* in a multibyte-character string. This function is typically used to transform the single-byte reverse-traversal loop to a multibyte loop. For example, suppose you have the following single-byte loop:

```
sb = sb0 + strlen(sb0);
for ( sb--; sb >= sb0; sb-- )
    /* Process single-byte character */
```

The following code fragment traverses the string with the **ifx_gl_mbsprev()** function instead of decrementing the pointer of the single-byte loop:

```
mb = mbs_start + strlen(mbs_start);
for ( mb = ifx_gl_mbsprev(mbs_start, mb) ; mb >= mbs_start;
    mb = ifx_gl_mbsprev(mbs_start, mb) )
    /* Process multibyte character */
```

### Return values

**gl_mchar_t ***
: A pointer to the first byte of the multibyte character immediately before *mb*.

**NULL** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns NULL and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
: The **mb* character or a character between *mbs_start* and *mb* is not a valid multibyte character.

**IFX_GL_EINVPTR**
: The *mb* value is less than or equal to *mbs_start*.

**IFX_GL_EINVAL**
: The function cannot determine whether the character before *mb is* a valid multibyte character because it would need to read beyond *mb*.

**Related concepts**:

"String traversal" on page 2-18

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"String-length determination" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mblen() function" on page 4-87

"The ifx_gl_mbsnext() function" on page 4-101

## The ifx_gl_mbsrchr() function

The **ifx_gl_mbsrchr()** function searches for the last occurrence of a character in a multibyte-character string.

## Syntax

```
#include <ifxgls.h>
...
gl_mchar_t *ifx_gl_mbsrchr(mbs, mbs_byte_length, mb, mb_byte_limit)
        gl_mchar_t *mbs;
        int mbs_byte_length;
        gl_mchar_t *mb;
        int mb_byte_limit;
```

*mbs* A pointer to the multibyte-character string in which the function searches for *mb*.

*mbs_byte_length*
The integer number of bytes in the *mbs* string. If *mbs_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs1* is a null-terminated string.

*mb* A pointer to the multibyte character to search for in *mbs*.

*mb_byte_limint*
The number of bytes to read from the *mb* character to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, this function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbsrchr()** function locates the last occurrence of mb in the multibyte string mbs.

## Return values

**gl_mchar_t ***
A pointer to the last occurrence of *mb* in *mbs*.

**NULL** Either *mb* was not found in *mbs* (the **ifx_gl_lc_errno()** error number is set to 0); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns NULL and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is an invalid multibyte character, or *mbs* contains an invalid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from *mb*, or the function cannot determine whether the last character of mbs is a valid multibyte character because it would need to read more than *mbs_byte_length* bytes from *mbs*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**IFX_GL_PARAMERR**
The *mbs_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**Related concepts**:

"Character-string termination" on page 2-22
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"Character searching" on page 2-20
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_wcschr() function" on page 4-120
"The ifx_gl_wcsrchr() function" on page 4-131

## The ifx_gl_mbsspn() function

The **ifx_gl_mbsspn()** function determines the length of a multibyte substring for a specified multibyte-character string.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbsspn(mbs1, mbs1_byte_length, mbs2, mbs2_byte_length)
       gl_mchar_t *mbs1;
       int mbs1_byte_length;
       gl_mchar_t *mbs2;
       int mbs2_byte_length;
```

*mbs1* A pointer to the multibyte string to check for a substring of characters found in *mbs2*.

*mbs1_byte_length*
The integer number of bytes in the *mbs1* string. If *mbs1_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs1* is a null-terminated string.

*mbs2* A pointer to the multibyte-character string whose characters the function searches for in *mbs1*.

*mbs2_byte_length*
The integer number of bytes in the *mbs2* string. If *mbs2_byte_length* is the value IFX_GL_NULL, the function assumes that *mbs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_mbsspn()** function returns the size of the initial substring of *mbs1* that contains only characters that match characters in *mbs2*. The initial substring begins at the first character of *mbs1*. Therefore, this size is the number of characters in *mbs1* before the first character that is not found in *mbs2*. For example, suppose you have the following two multibyte-characters strings, *mbs1* and *mbs2*. For example, suppose you have the following two multibyte-characters strings, mbs1 and mbs2:

```
mbs1 = "A1A2B1B2B3C1C2D1D2D3A1A2E1F1F2A1A2G1D1D2D3";
mbs2 = "B1B2B3D1D2D3A1A2C2C2";
```

With these multibyte strings, the following call to the **ifx_gl_mbsspn()** function returns 5:

```
ifx_gl_mbsspn(mbs1, bytelen1, mbs2, bytelen2)
```

The first five characters of `mbs1` are in `mbs2`. The sixth character in `mbs1` is $E^1$, which is not a character that matches one of the characters in `mbs2`.

### Return values

**>=0** The number of characters in *mbs1* before the first character for which *mbs1* and *mbs2* differ.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *mbs1_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *mbs2_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *mbs1_byte_length* is equal to IFX_GL_NULL, and *mbs2_byte_length* is greater than or equal to 0; or *mbs1_byte_length* is greater than or equal to 0, and *mbs2_byte_length* is equal to IFX_GL_NULL.

**IFX_GL_EILSEQ**
Either *mbs1* or *mbs2* contains an invalid multibyte character.

**IFX_GL_EINVAL**
Either the function cannot determine whether the last character of *mbs1* is a valid multibyte character because it would need to read more than *mbs1_byte_length* bytes from *mbs1*, or the function cannot determine whether the last character of *mbs2* is a valid multibyte character because it would need to read more than *mbs2_byte_length* bytes from *mbs2*.

**Related concepts**:

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_wcscspn() function" on page 4-124

"The ifx_gl_wcsspn() function" on page 4-132

## The ifx_gl_mbstowcs() function

The **ifx_gl_mbstowcs()** function converts a multibyte character string to its wide-character representation.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbstowcs(wcs, mbs, mbs_byte_length, char_limit)
```

```
gl_wchar_t *wcs;
gl_mchar_t *mbs;
int mbs_byte_length;
int char_limit;
```

*wcs* A pointer to the wide-character string that contains the wide-character equivalent of *mbs*.

*mbs* A pointer to the multibyte-character string to convert to the *wcs* wide-character string.

*mbs_byte_length*
The integer number of bytes in the *mbs* string. If *mbs_byte_length* is the value IFX_GL_NULL, the function assumes that mbs is a null-terminated string.

**char_limit**
The maximum number of multibyte characters to read from mbs.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbstowcs()** function converts mbs into its wide-character representation and stores the result in the location that *wcs* references. The function reads no more than *char_limit* characters from mbs and writes them to *wcs*.

If *mbs_byte_length* is equal to IFX_GL_NULL, the function null-terminates the string in *wcs*. Any other value of *mbs_byte_length* means that the function does not null-terminate the resulting wide-character string.

## Return values

**>=0** The number of characters read from *mbs* and written to *wcs*, not counting any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
The *mbs_byte_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_EILSEQ**
The **mbs* value contains an invalid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether the last character of *mbs* is a valid multibyte character because it would need to read more than *mbs_byte_length* bytes from *mbs*.

**Related concepts**:

"Character-string termination" on page 2-22

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbtowc() function"

"The ifx_gl_wcstombs() function" on page 4-134

# The ifx_gl_mbtowc() function

The **ifx_gl_mbtowc()** function converts one multibyte character to its wide-character representation.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_mbtowc(wc, mb, mb_byte_limit)
       gl_wchar_t *wc;
       gl_mchar_t *mb;
       int mb_byte_limit;
```

*wc* A pointer to the wide-character string that contains the wide-character equivalent of *mb*.

*mb* A pointer to the multibyte-character to convert to the *wc* wide-character .

*mb_byte_length*
The integer number of bytes to read from *mb* to try to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, the function reads as many bytes as necessary from *mb* to form a complete character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_mbtowc()** function converts the multibyte character *mb* into its wide-character representation and stores the result in the wide character that wc references.

## Return values

**>=0** The number of bytes read from mb.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *mb* is a valid multibyte character

because it would need to read more than *mb_byte_limit* bytes from *mb*. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Multibyte-character termination" on page 2-23

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbstowcs() function" on page 4-110

"The ifx_gl_wcstombs() function" on page 4-134

"The ifx_gl_wctomb() function" on page 4-136

# The ifx_gl_putmb() function

The **ifx_gl_putmb()** function puts a single multibyte character in a user-defined location.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_putmb(mb, mb_byte_limit, funcp, v, bytes_put)
        gl_mchar_t *mb;
        int mb_byte_limit;
        int (*funcp)(gl_mchar_t byte, void *v);
        void *v;
        int *bytes_put;
```

*mb* A pointer to the multibyte character whose bytes the *funcp* function writes to a specified location.

*mb_byte_limit*
The number of bytes in *mb* that the **ifx_gl_putmb()** function reads when it tries to form a complete multibyte character. If *mb_byte_limit* is IFX_GL_NO_LIMIT, this function reads as many bytes as necessary from *mb* to form a complete character.

*funcp* A pointer to a function that you define to specify the location at which to write the multibyte character.

*v* An argument to the user-defined function, which is a parameter to **ifx_gl_putmb()**. Use *v* to specify where the character should go; for example, stdout, stderr, a file, and other locations.

*bytes_put*

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_putmb()** function calls a function that you define to put the bytes that form one multibyte character in a specified location. This multibyte character is read from the *mb* buffer.

The *funcp* argument is a pointer to a function that you must define as follows:

```
int funcp(gl_mchar_t byte, void *v)
```

On success, *funcp* must return 0. On failure, *funcp* must return -1. The number of bytes that *funcp* successfully puts is returned in *bytes_put* (even when *funcp* fails).

### Return values

**0** The function was successful.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The *funcp* function cannot determine whether *mb* is a valid multibyte character because it would need to read more than *mb_byte_limit* bytes from mb. If *mb_byte_limit* is less than or equal to 0, this function always returns this error.

**IFX_GL_EIO**
The *funcp* function returned -1 when first called.

**Related concepts**:

"Multibyte-character termination" on page 2-23

"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_getmb() function" on page 4-51

"The ifx_gl_lc_errno() function" on page 4-85

## The ifx_gl_tomlower() function

The **ifx_gl_tomlower()** function converts an uppercase multibyte character to its lowercase equivalent.

### Syntax

```
#include <ifxgls.h>
...
unsigned short ifx_gl_tomlower(dst_mb, src_mb, src_mb_byte_limit)
       gl_mchar_t *dst_mb;
       gl_mchar_t *src_mb;
       int src_mb_byte_limit;
```

*dst_mb* A pointer to the destination character, which holds the case conversion for the *src_mb* character.

*src_mb* A pointer to the source multibyte character that you want to convert to its lowercase equivalent.

*src_mb_byte_limit*
The integer number of bytes to read from *src_mb* to try to form a complete multibyte character. If *src_mb_byte_limit* is IFX_GL_NO_LIMIT, this function reads as many bytes as necessary from *src_mb* to form a complete multibyte character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_tomlower()** function obtains the lowercase equivalent of the *src_mb* source multibyte character and stores it in the *dst_mb* destination buffer. If the locale does not define a lowercase equivalent for *src_mb*, **ifx_gl_tomlower()** copies *src_mb* to *dst_mb* unchanged.

For a multibyte-character string, the size of the lowercase multibyte string might not equal the size of the uppercase string. Therefore, to perform case conversion on multibyte characters, you must take the following special processing steps:

- Determine whether you need to allocate a separate destination buffer.
  If a destination buffer is needed, determine its size.
- Determine the number of bytes that have been read and written in the case-conversion process.

The **ifx_gl_tomlower()** function returns an unsigned short integer that encodes the number of bytes read from *src_mb* and the number of bytes written to *dst_mb*. The IBM Informix GLS library provides the following macros to obtain this information from the return value.

**IFX_GL_CASE_CONV_SRC_BYTES()**
The number of bytes read from the source string.

**IFX_GL_CASE_CONV_DST_BYTES()**
The number of bytes written to the destination buffer.

## Return values

**>0** An unsigned short integer that encodes the number of bytes read from *src_mb* and the number of bytes written to *dst_mb*.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **src_mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *src_mb* is a valid multibyte character because it would need to read more than *src_mb_byte_limit* bytes from *src_mb*. If *src_mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Case conversion" on page 2-8
"Case conversion for multibyte characters" on page 2-9
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_case_conv_outbuflen() function" on page 4-4
"The ifx_gl_ismlower() function" on page 4-62
"The ifx_gl_ismupper() function" on page 4-68
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mb_loc_max() function" on page 4-86
"The ifx_gl_tomupper() function"
"The ifx_gl_towlower() function" on page 4-117

# The ifx_gl_tomupper() function

The **ifx_gl_tomupper()** function converts a lowercase multibyte character to its uppercase equivalent.

## Syntax

```
#include <ifxgls.h>
...
unsigned short ifx_gl_tomupper(dst_mb, src_mb, src_mb_byte_limit)
       gl_mchar_t *dst_mb;
       gl_mchar_t *src_mb;
       int src_mb_byte_limit;
```

*dst_mb* A pointer to the destination character, which holds the case conversion for the *src_mb* character.

*src_mb* A pointer to the source multibyte character that you want to convert to its uppercase equivalent.

*src_mb_byte_limit*
The integer number of bytes to read from *src_mb* to try to form a complete multibyte character. If *src_mb_byte_limit* is IFX_GL_NO_LIMIT, this function reads as many bytes as necessary from *src_mb* to form a complete multibyte character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_tomupper()** function obtains the uppercase equivalent of the *src_mb* source multibyte character and stores it in the *dst_mb* destination buffer. If the locale does not define an uppercase equivalent for *src_mb*, **ifx_gl_tomupper()** copies *src_mb* to *dst_mb* unchanged.

For a multibyte-character string, the size of the uppercase multibyte string might not equal the size of the uppercase string. Therefore, to perform case conversion on multibyte characters, you must take the following special processing steps:

- Determine whether you need to allocate a separate destination buffer.
  If a destination buffer is needed, determine its size.

- Determine the number of bytes that have been read and written in the case-conversion process.

The **ifx_gl_tomupper()** function returns an unsigned short integer that encodes the number of bytes read from *src_mb* and the number of bytes written to *dst_mb*. The IBM Informix GLS library provides the following macros to obtain this information from the return value.

**IFX_GL_CASE_CONV_SRC_BYTES()**
The number of bytes read from the source string.

**IFX_GL_CASE_CONV_DST_BYTES()**
The number of bytes written to the destination buffer.

### Return values

**>0** An unsigned short integer that encodes the number of bytes read from *src_mb* and the number of bytes written to *dst_mb*.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_EILSEQ**
The **src_mb* value is not a valid multibyte character.

**IFX_GL_EINVAL**
The function cannot determine whether *src_mb* is a valid multibyte character because it would need to read more than *src_mb_byte_limit* bytes from *src_mb*. If *src_mb_byte_limit* is less than or equal to 0, this function always returns this error.

**Related concepts**:

"Case conversion" on page 2-8
"Case conversion for multibyte characters" on page 2-9
"Multibyte-character termination" on page 2-23
"Keep multibyte strings consistent" on page 2-26

**Related reference**:

"The ifx_gl_case_conv_outbuflen() function" on page 4-4
"The ifx_gl_ismlower() function" on page 4-62
"The ifx_gl_ismupper() function" on page 4-68
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mb_loc_max() function" on page 4-86
"The ifx_gl_tomlower() function" on page 4-114
"The ifx_gl_towupper() function" on page 4-118

## The ifx_gl_towlower() function

The **ifx_gl_towlower()** function converts an uppercase wide character to its lowercase equivalent.

### Syntax

```
#include <ifxgls.h>
...
unsigned short ifx_gl_towlower(src_wc)
       gl_mchar_t *src_wc;
```

*src_wc* A pointer to the source wide character that you want to convert to its lowercase equivalent.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_towlower()** function returns the lowercase equivalent of the *src_wc* source wide character. If the locale does not define a lowercase equivalent for *src_wc*, **ifx_gl_towlower()** returns *src_wc* unchanged.

### Return values

**>0** An unsigned short integer that represents the lowercase equivalent of *src_wc*.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **src_wc* value is not a valid wide character.

**Related concepts**:

"Case conversion" on page 2-8

**Related reference**:

"The ifx_gl_case_conv_outbuflen() function" on page 4-4
"The ifx_gl_iswlower() function" on page 4-78
"The ifx_gl_iswupper() function" on page 4-83
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mb_loc_max() function" on page 4-86
"The ifx_gl_tomlower() function" on page 4-114
"The ifx_gl_towupper() function"

## The ifx_gl_towupper() function

The **ifx_gl_towupper()** function converts a lowercase wide character to its uppercase equivalent.

### Syntax

```
#include <ifxgls.h>
...
unsigned short ifx_gl_towupper(src_wc)
       gl_mchar_t *src_wc;
```

*src_wc* A pointer to the source wide character that you want to convert to its uppercase equivalent.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_towupper()** function returns the uppercase equivalent of the *src_wc* source wide character. If the locale does not define an uppercase equivalent for *src_wc*, **ifx_gl_towupper()** returns *src_wc* unchanged.

### Return values

**>0** An unsigned short integer that represents the uppercase equivalent of *src_wc*.

**0** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

This function does not return a unique value to indicate an error. If an error occurred, the function returns 0 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **src_wc* value is not a valid wide character.

**Related concepts**:

"Case conversion" on page 2-8

**Related reference**:

"The ifx_gl_case_conv_outbuflen() function" on page 4-4
"The ifx_gl_iswlower() function" on page 4-78
"The ifx_gl_iswupper() function" on page 4-83
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mb_loc_max() function" on page 4-86
"The ifx_gl_tomupper() function" on page 4-116
"The ifx_gl_towlower() function" on page 4-117

## The ifx_gl_wcscat() function

The **ifx_gl_wcscat()** function concatenates two wide-character strings.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcscat(wcs1, wcs1_char_length, wcs2, wcs2_char_length)
       gl_wchar_t *wcs1;
       int wcs1_char_length;
       gl_wchar_t *wcs2;
       int wcs2_char_length;
```

*wcs1* A pointer to the wide-character string to which the function concatenates *wcs2*.

*wcs1_char_length*
The integer number of characters in the *wcs1* string. If *wcs1_char_length* is the value IFX_GL_NULL, the function assumes that *wcs1* is a null-terminated string.

*wcs2* A pointer to the wide-character string to concatenate onto *wcs1*.

*wcs2_char_length*
The integer number of characters in the *wcs2* string. If *wcs2_char_length* is the value IFX_GL_NULL, the function assumes that *wcs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_wcscat()** function appends a copy of *wcs2* to the end of *wcs1*. If *wcs1* and *wcs2* overlap, the results of this function are undefined.

### Return values

**>=0** The number of characters in the concatenated string, not including any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *wcs1_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *wcs2_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *wcs1_char_length* is equal to IFX_GL_NULL, and *wcs2_char_length* is greater than or equal to 0; or *wcs1_char_length* is greater than or equal to 0, and *wcs2_char_length* is equal to IFX_GL_NULL.

**Related concepts**:

"Concatenation" on page 2-19

"Character-string termination" on page 2-22

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbscat() function" on page 4-88

"The ifx_gl_mbsncat() function" on page 4-98

"The ifx_gl_wcsncat() function" on page 4-126

## The ifx_gl_wcschr() function

The **ifx_gl_wcschr()** function searches for the first occurrence of a character in a wide-character string.

### Syntax

```
#include <ifxgls.h>
...
gl_wchar_t *ifx_gl_wcschr(wcs, wcs_char_length, wc)
        gl_wchar_t *wcs;
        int wcs_char_length;
        gl_wchar_t wc;
```

*wcs* A pointer to the wide-character string in which the function searches for *wc*.

*wcs_char_length*
The integer number of characters in the wcs string. If wcs_char_length is the value IFX_GL_NULL, the function assumes that wcs is a null-terminated string.

*wc* A pointer to the wide character to search for in *wcs*.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_wcschr()** function locates the first occurrence of *wc* in the wide-character string *wcs*.

### Return values

**gl_wchar_t ***
A pointer to the first occurrence of *wc* in *wcs*.

**NULL** Either *wc* was not found in *wcs* (the **ifx_gl_lc_errno()** error number is set to 0); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns NULL and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_PARAMERR**
The *wcs_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**Related concepts**:

"Character-string termination" on page 2-22

**Related reference**:

"Character searching" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbschr() function" on page 4-89

"The ifx_gl_wcsrchr() function" on page 4-131

## The ifx_gl_wcscoll() function

The **ifx_gl_wcscoll()** function uses locale-specific order to compare two wide-character strings.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcscoll(wcs1, wcs1_char_length, wcs2, wcs2_char_length)
       gl_wchar_t *wcs1;
       int wcs1_char_length;
       gl_wchar_t *wcs2;
       int wcs2_char_length;
```

*wcs1* A pointer to the wide-character string to compare with *wcs2*.

*wcs1_char_length*
The integer number of characters in the *wcs1* string. If *wcs1_char_length* is the value IFX_GL_NULL, the function assumes that *wcs1* is a null-terminated string.

*wcs2* A pointer to the wide-character string to compare with *wcs1*.

*wcs2_char_length*
The integer number of characters in the *wcs2* string. If *wcs2_char_length* is the value IFX_GL_NULL, the function assumes that *wcs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_wcscoll()** function uses locale-specific order to compare the wide-character strings *wcs1* and *wcs2*.

## Locale information

The LC_COLLATE category of the current locale affects the behavior of this function because it defines the locale-specific order.

## Return values

**<0** The *wcs1* string is less than the *wcs2* string (*wcs1* < *wcs2*); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

**=0** The *wcs1* string is equal to the *wcs* string (*wcs1* = *wcs*).

**>0** The *wcs1* string is greater than the *wcs* string (*wcs1* > *wcs*).

## Errors

This function does not return a unique value to indicate an error. If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *wcs1_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *wcs2_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *wcs1_char_length* is equal to IFX_GL_NULL, and *wcs2_char_length* is

greater than or equal to 0; or *wcs1_char_length* is greater than or equal to 0, and *wcs2_char_length* is equal to IFX_GL_NULL.

**IFX_GL_EILSEQ**
Either *wcs1* or *wcs2* contains an invalid wide character.

**IFX_GL_ENOMEM**
Not enough memory is available to complete the operation.

**Related concepts**:

"Character/string comparison and sorting" on page 2-20

"Character-string termination" on page 2-22

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbscoll() function" on page 4-91

# The ifx_gl_wcscpy() function

The **ifx_gl_wcscpy()** function copies a wide-character string.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcscpy(wcs1, wcs2, wcs2_char_length)
       gl_wchar_t *wcs1;
       gl_wchar_t *wcs2;
       int wcs2_char_length;
```

*wcs1* A pointer to the location where the function copies *wcs2*.

*wcs2* A pointer to the wide-character string to copy to *wcs1*.

*wcs2_char_length*
The integer number of characters in the *wcs2* string. If *wcs2_char_length* is the value IFX_GL_NULL, the function assumes that *wcs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_wcscpy()** function copies the wide-character string *wcs2* to the location that *wcs1* references. If *wcs1* and *wcs2* overlap, the results of this function are undefined.

## Return values

**>=0** The number of bytes in the copied string, not including any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_PARAMERR**
The *wcs2_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**Related concepts**:

"Character-string termination" on page 2-22

**Related reference**:

"String copying" on page 2-19
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbscpy() function" on page 4-92
"The ifx_gl_mbsncpy() function" on page 4-99
"The ifx_gl_wcsncpy() function" on page 4-128

# The ifx_gl_wcscspn() function

The **ifx_gl_wcscspn()** function determines the length of a complementary wide-character substring for a wide-character string.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcscspn(wcs1, wcs1_char_length, wcs2, wcs2_char_length)
       gl_wchar_t *wcs1;
       int wcs1_char_length;
       gl_wchar_t *wcs2;
       int wcs2_char_length;
```

*wcs1* A pointer to the wide-character string to check for the complementary substring of characters in *wcs2*.

*wcs1_char_length*
The integer number of characters in the *wcs1* string. If *wcs1_char_length* is the value IFX_GL_NULL, the function assumes that *wcs1* is a null-terminated string. For more information about wide-character lengths, see "Character-string termination" on page 2-22.

*wcs2* A pointer to the string of wide-characters to search in *wcs1*.

*wcs2_char_length*
The integer number of characters in the *wcs2* string. If *wcs2_char_length* is the value IFX_GL_NULL, the function assumes that *wcs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_wcscspn()** function returns the size of the initial substring of *wcs1* that does not contain any characters that match characters in *wcs2*. The initial substring begins at the first character of *wcs1*. Therefore, this size is the number of characters in *wcs1* before the first character that the *wcs1* and *wcs2* strings have in common. The string of unmatched characters is called the *complementary string*.

For example, suppose you have the following two wide-character strings, `wcs1` and `wcs2`:

```
wcs1 = "A1A2B1B2C1C2D1D2A1A2E1E2F1F2A1A2G1G2D1D2";
wcs2 = "F1F2G1D1D2";
```

With these wide-character strings, the following call to the **ifx_gl_wcscspn()** function returns 3:

```
ifx_gl_wcscspn(wcs1, charlen1, wcs2, charlen2)
```

The first three characters of `wcs1` are not in `wcs2`. The fourth character in `wcs1` is $D^1D^2$, which is in `wcs2` also.

### Return values

**>=0** The number of characters in the initial substring of *wcs1* that consist entirely of wide characters not in the string *wcs2*.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns NULL and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *wcs1_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *wcs2_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *wcs1_char_length* is equal to IFX_GL_NULL, and *wcs2_char_length* is greater than or equal to 0; or *wcs1_char_length* is greater than or equal to 0, and *wcs2_char_length* is equal to IFX_GL_NULL.

**Related concepts**:

"Character-string termination" on page 2-22

**Related reference**:

"String-length determination" on page 2-20
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbscspn() function" on page 4-93
"The ifx_gl_mbsspn() function" on page 4-109
"The ifx_gl_wcsspn() function" on page 4-132

## The ifx_gl_wcslen() function

The **ifx_gl_wcslen()** function determines the number of characters in a wide-character string.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcslen(wcs)
       gl_wchar_t *wcs;
```

*wcs* A pointer to the wide-character string whose length the function determines.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_wcslen()** function computes the number of wide characters in the wide-character string to which *wcs* points, not including the null-terminating wide-character code. The length that **ifx_gl_wcslen()** returns includes any trailing white space. The trailing-space characters are all characters that the blank character class of the current locale defines.

For example, the following call to **ifx_gl_wcslen()** return a value of 7 (where $A^1A^2$, $B^1B^2$, and $C^1C^2$ are wide characters of 2 bytes each, and $s^1s^2$ is the wide-character blank space):

```
ifx_gl_wcslen("A1A2B1B2s1s2C1C2s1s2s1s2s1s2")
```

Use the ifx_gl_wcntslen() function if you want the length without trailing white space.

### Return values

**>=0** The number of characters in the *wcs* string, not including any null terminator but including any trailing white space.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wcs* value contains an invalid wide character.

**Related concepts**:

"Character classification" on page 2-5

**Related reference**:

"String-length determination" on page 2-20
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbslen() function" on page 4-95
"The ifx_gl_mbsntsbytes() function" on page 4-102
"The ifx_gl_mbsntslen() function" on page 4-103
"The ifx_gl_wcsntslen() function" on page 4-129

## The ifx_gl_wcsncat() function

The **ifx_gl_wcsncat()** function concatenates a specified number of wide characters in one wide-character string to another.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcsncat(wcs1, wcs1_char_length, wcs2, wcs2_char_length, char_limit)
       gl_wchar_t *wcs1;
       int wcs1_char_length;
       gl_wchar_t *wcs2;
       int wcs2_char_length;
       int char_limit;
```

*wcs1* A pointer to the wide-character string to which the function concatenates *wcs2*.

*wcs1_char_length*
The integer number of characters in the *wcs1* string. If *wcs1_char_length* is the value IFX_GL_NULL, the function assumes that *wcs1* is a null-terminated string.

*wcs2* A pointer to the wide-character string to concatenate to *wcs1*.

*wcs2_char_length*
The integer number of characters in the *wcs2* string. If *wcs2_char_length* is the value IFX_GL_NULL, the function assumes that *wcs2* is a null-terminated string.

*char_limit*
The maximum number of wide characters to read from *wcs2*.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_wcsncat()** function appends *wcs2* to the end of *wcs1*. If *wcs1* and *wcs2* overlap, the results of this function are undefined. The function reads no more than *char_limit* characters from *wcs2* and writes them to *wcs1*.

## Return values

**>=0** The number of characters in the concatenated string, not including any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *wcs1_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *wcs2_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *wcs1_char_length* is equal to IFX_GL_NULL, and *wcs2_char_length* is greater than or equal to 0; or *wcs1_char_length* is greater than or equal to 0, and *wcs2_char_length* is equal to IFX_GL_NULL.

**Related concepts**:

"Concatenation" on page 2-19

"Character-string termination" on page 2-22

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbscat() function" on page 4-88

"The ifx_gl_mbsncat() function" on page 4-98

"The ifx_gl_wcscat() function" on page 4-119

# The ifx_gl_wcsncpy() function

The **ifx_gl_wcsncpy()** function copies a specified number of wide characters from a wide-character string.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcsncpy(wcs1, wcs2, wcs2_char_length, char_limit)
       gl_wchar_t *wcs1;
       gl_wchar_t *wcs2;
       int wcs2_char_length;
       int char_limit;
```

*wcs1* A pointer to the location where the function copies *wcs2*.

*wcs2* A pointer to the wide-character string to copy to *wcs1*.

*wcs2_char_length*
The integer number of characters in the *wcs2* string. If *wcs2_char_length* is the value IFX_GL_NULL, the function assumes that *wcs2* is a null-terminated string.

*char_limit*
The maximum number of wide characters to read from *wcs2*.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_wcsncpy()** function copies *wcs2* to the location that *wcs1* references. The function reads no more than *char_limit* characters from *wcs2* and writes them to *wcs1*. If *wcs1* and *wcs2* overlap, the results of this function are undefined.

## Return values

**>=0** The number of bytes in the copied string, not including any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_PARAMERR**
The *wcs2_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**Related concepts**:

"Character-string termination" on page 2-22

**Related reference**:

"String copying" on page 2-19
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbscpy() function" on page 4-92
"The ifx_gl_mbsncpy() function" on page 4-99
"The ifx_gl_wcscpy() function" on page 4-123

# The ifx_gl_wcsntslen() function

The **ifx_gl_wcsntslen()** function determines the number of characters in a wide-character string.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcsntslen(wcs, wcs_char_length)
       gl_wchar_t *wcs;
       int wcs_char_length;
```

*wcs* A pointer to the wide-character string whose length the function determines.

*wcs_char_length*
The integer number of characters in the *wcs* string. If *wcs_char_length* is the value IFX_GL_NULL, the function assumes that *wcs* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_wcsntslen()** function returns the number of characters in *wcs*, not including the trailing-space characters. The trailing-space characters are all characters that the blank character class of the current locale defines.

Space characters that are embedded in the string at any place except the end of the string are included in the count. For example, the following call to **ifx_gl_wcsntslen()** return a value of 4 (the number of wide characters in the string "$A^1A^2B^1B^2s^1s^2C^1C^2$", where $A^1A^2$, $B^1B^2$, and $C^1C^2$ are wide characters of 2 bytes each, and $s^1s^2$ is the wide-character blank space):

```
ifx_gl_wcsntslen("A¹A²B¹B²s¹s²C¹C²s¹s²s¹s²s¹s²", wcs_char_length)
```

Use the **ifx_gl_wcntslen()** function if you want the length without trailing white space.

## Return values

**>=0** The number of characters in the *wcs* string, not including any trailing space.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_PARAMERR**
The *wcs_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**Related concepts**:

"Character classification" on page 2-5

"Character-string termination" on page 2-22

**Related reference**:

"String-length determination" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbsntsbytes() function" on page 4-102

"The ifx_gl_mbsntslen() function" on page 4-103

"The ifx_gl_wcslen() function" on page 4-125

## The ifx_gl_wcspbrk() function

The **ifx_gl_wcspbrk()** function searches for any wide character from one wide-character string in another wide-character string.

### Syntax

```
#include <ifxgls.h>
...
gl_wchar_t *ifx_gl_wcspbrk(wcs1, wcs1_char_length, wcs2, wcs2_char_length)
        gl_wchar_t *wcs1;
        int wcs1_char_length;
        gl_wchar_t *wcs2;
        int wcs2_char_length;
```

*wcs1* A pointer to the wide-character string to search for any character in *wcs2*.

*wcs1_char_length*
The integer number of characters in the *wcs1* string. If *wcs1_char_length* is the value IFX_GL_NULL, the function assumes that *wcs1* is a null-terminated string.

*wcs2* A pointer to the wide-character string whose characters the function searches for in *wcs1*.

*wcs2_char_length*
The integer number of characters in the *wcs2* string. If *wcs2_char_length* is the value IFX_GL_NULL, the function assumes that *wcs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_wcspbrk()** function searches for the first occurrence of any wide character from the string *wcs2* in the wide-character string *wcs1*.

### Return values

**gl_wchar_t ***
A pointer to the first byte of the first occurrence in *wcs1* of any character from *wcs2*.

**NULL** Either no character in *wcs2* was found in *wcs1* (the **ifx_gl_lc_errno()** error number is set to 0); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns NULL and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *wcs1_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *wcs2_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *wcs1_char_length* is equal to IFX_GL_NULL, and *wcs2_char_length* is greater than or equal to 0; or *wcs1_char_length* is greater than or equal to 0, and *wcs2_char_length* is equal to IFX_GL_NULL.

**Related concepts**:

"Character-string termination" on page 2-22

**Related reference**:

"Character searching" on page 2-20
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbschr() function" on page 4-89
"The ifx_gl_mbspbrk() function" on page 4-105
"The ifx_gl_mbsrchr() function" on page 4-107
"The ifx_gl_wcschr() function" on page 4-120
"The ifx_gl_wcsrchr() function"

## The ifx_gl_wcsrchr() function

The **ifx_gl_wcsrchr()** function searches for the last occurrence of a character in a wide-character string.

### Syntax

```
#include <ifxgls.h>
...
gl_wchar_t *ifx_gl_wcsrchr(wcs, wcs_char_length, wc)
        gl_wchar_t *wcs;
        int wcs_char_length;
        gl_wchar_t wc;
```

*wcs* A pointer to the wide-character string in which the function searches for *wc*.

*wcs_char_length*
The integer number of characters in the *wcs* string. If *wcs_char_length* is the value IFX_GL_NULL, the function assumes that *wcs* is a null-terminated string.

*wc* A pointer to the wide character to search for in *wcs*.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_wcsrchr()** function locates the last occurrence of *wc* in the wide character string *wcs*.

### Return values

**gl_wchar_t ***
A pointer to the last occurrence of *wc* in *wcs*.

**NULL** Either *wc* was not found in *wcs* (the **ifx_gl_lc_errno()** error number is set to 0); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns NULL and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_PARAMERR**
The *wcs_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**Related concepts**:

"Character-string termination" on page 2-22

**Related reference**:

"Character searching" on page 2-20
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbschr() function" on page 4-89
"The ifx_gl_mbsrchr() function" on page 4-107
"The ifx_gl_wcschr() function" on page 4-120

## The ifx_gl_wcsspn() function

The **ifx_gl_wcsspn()** function determines the length of a wide-character substring for a specified wide-character string.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcsspn(wcs1, wcs1_char_length, wcs2, wcs2_char_length)
       l_wchar_t *wcs1;
       int wcs1_char_length;
       gl_wchar_t *wcs2;
       int wcs2_char_length;
```

*wcs1* A pointer to the wide-character string to check for a substring of characters found in *wcs2*.

*wcs1_char_length*
The integer number of characters in the *wcs1* string. If *wcs1_char_length* is the value IFX_GL_NULL, the function assumes that *wcs1* is a null-terminated string.

*wcs2* A pointer to the wide-character string whose characters the function searches for in *wcs1*.

*wcs2_char_length*
The integer number of characters in the *wcs2* string. If *wcs2_char_length* is the value IFX_GL_NULL, the function assumes that *wcs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_wcsspn()** function returns the size of the initial substring of *wcs1* that contains only characters that match characters in *wcs2*. The initial substring begins at the first character of *wcs1*. Therefore, this size is the number of characters in *wcs1* before the first character that is not found in *wcs2*.

For example, suppose you have the following two wide-character strings, `wcs1` and `wcs2`:

```
wcs1 = "A1A2B1B2C1C2D1D2A1A2E1E2F1F2A1A2G1G2D1D2";
wcs2 = "B1B2D1D2A1A2C1C2";
```

With these wide-character strings, the following call to the **ifx_gl_wcsspn()** function returns 5:

```
ifx_gl_wcsspn(wcs1, charlen1, wcs2, charlen2)
```

The first five characters of `wcs1` are in `wcs2`. The sixth character in `wcs1` is $E^1E^{2,}$ which is not a character that matches one of the characters in `wcs2`.

## Return values

**>=0** The number of characters in the initial substring of *wcs1* that consist entirely of wide characters in the string *wcs2*.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *wcs1_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *wcs2_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *wcs1_char_length* is equal to IFX_GL_NULL, and *wcs2_char_length* is greater than or equal to 0; or *wcs1_char_length* is greater than or equal to 0, and *wcs2_char_length* is equal to IFX_GL_NULL.

**Related concepts**:

"Character-string termination" on page 2-22

**Related reference**:

"String-length determination" on page 2-20

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbscspn() function" on page 4-93

"The ifx_gl_mbsspn() function" on page 4-109

"The ifx_gl_wcscspn() function" on page 4-124

# The ifx_gl_wcstombs() function

The **ifx_gl_wcstombs()** function converts a wide-character string to its multibyte representation.

## Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wcstombs(mbs, wcs, wcs_char_length, mbs_byte_limit)
       gl_mchar_t *mbs;
       gl_wchar_t *wcs;
       int wcs_char_length;
       int mbs_byte_limit;
```

*mbs* A pointer to the multibyte-character string that contains the multibyte equivalent for *wcs*.

*wcs* A pointer to the wide-character string to convert to the *mbs* multibyte string.

*wcs_char_length*
The integer number of characters in the *wcs* string. If *wcs_char_length* is the value IFX_GL_NULL, the function assumes that *wcs* is a null-terminated string.

*mbs_byte_limit*
The integer number of bytes to read from *mbs*.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_wcstombs()** function converts the wide-character string *wcs* to its multibyte representation and stores the result in the location that *mbs* references.

The function writes at most *mbs_byte_limit* bytes to *mbs*. If a particular character would cause more than *mbs_byte_limit* bytes to be written to *mbs*, no part of that character is written to *mbs*. In this case, fewer than *mbs_byte_limit* bytes are written to *mbs*, but mbs is still considered full.

If *wcs_char_length* is equal to IFX_GL_NULL, the function null-terminates *mbs*. Any other value of *wcs_char_length* means that the function does not null-terminate the resulting string.

### Return values

**>=0** The number of characters that were written to *mbs*, not including any null terminator.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
The *wcs_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_EILSEQ**
The **wcs* value contains an invalid wide character.

**Related concepts**:

"Character-string termination" on page 2-22

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mbstowcs() function" on page 4-110

"The ifx_gl_mbtowc() function" on page 4-112

"The ifx_gl_wctomb() function" on page 4-136

## The ifx_gl_wcswcs() function

The **ifx_gl_wcswcs()** function searches for a specified substring within a wide-character string.

### Syntax

```
#include <ifxgls.h>
...
gl_wchar_t *ifx_gl_wcswcs(wcs1, wcs1_char_length, wcs2, wcs2_char_length)
        gl_wchar_t *wcs1;
        int wcs1_char_length;
        gl_wchar_t *wcs2;
        int wcs2_char_length;
```

*wcs1* A pointer to the wide-character string to search for the substring *wcs2*.

*wcs1_char_length*
The integer number of characters in the *wcs1* string. If *wcs1_char_length* is the value IFX_GL_NULL, the function assumes that *wcs1* is a null-terminated string.

*wcs2* A pointer to the wide-character substring to search for in *wcs1*.

*wcs2_char_length*
The integer number of characters in the *wcs2* string. If *wcs2_char_length* is the value IFX_GL_NULL, the function assumes that *wcs2* is a null-terminated string.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

### Usage

The **ifx_gl_wcswcs()** function searches for the first occurrence of the wide-character string *wcs2* in the wide-character string *wcs1*.

### Return values

**gl_wchar_t ***
A pointer to the first byte of the first occurrence of *wcs2* in *wcs1*.

**NULL** Either *wcs2* was not found in *wcs1* (the **ifx_gl_lc_errno()** error number is set to 0); or the function was not successful, and the error number is set to indicate the cause. See the Errors section.

### Errors

If an error occurred, this function returns NULL and sets the **ifx_gl_lc_errno()** error number to one of the following values.

**IFX_GL_PARAMERR**
Either *wcs1_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0, or *wcs2_char_length* is not equal to IFX_GL_NULL and is not greater than or equal to 0.

**IFX_GL_TERMMISMAT**
Either *wcs1_char_length* is equal to IFX_GL_NULL, and *wcs2_char_length* is greater than or equal to 0; or *wcs1_char_length* is greater than or equal to 0, and *wcs2_char_length* is equal to IFX_GL_NULL.

**Related concepts**:

"Character-string termination" on page 2-22

**Related reference**:

"Character searching" on page 2-20
"The ifx_gl_lc_errno() function" on page 4-85
"The ifx_gl_mbschr() function" on page 4-89
"The ifx_gl_mbsmbs() function" on page 4-96
"The ifx_gl_mbsrchr() function" on page 4-107
"The ifx_gl_wcschr() function" on page 4-120
"The ifx_gl_wcsrchr() function" on page 4-131

## The ifx_gl_wctomb() function

The **ifx_gl_wctomb()** function converts one wide character to its multibyte representation.

### Syntax

```
#include <ifxgls.h>
...
int ifx_gl_wctomb(mb, wc)
       gl_mchar_t *mb;
       gl_wchar_t wc;
```

*mb* A pointer to the multibyte character that contains the multibyte equivalent for *wc*.

*wc* A pointer to the wide character to convert to the *mb* multibyte character.

| Valid in client application | Valid in DataBlade UDR |
|---|---|
| Yes | Yes |

## Usage

The **ifx_gl_wctomb()** function converts the wide character in *wc* to its multibyte representation and stores the result in consecutive bytes that start at *mb*. This function assumes that *mb* contains enough space to hold the multibyte representation of *wc*. You can use either of the following methods to determine the number of bytes that will be written to *mb*:

- The function **ifx_gl_mb_loc_max()** calculates the maximum number of bytes in the multibyte representation for any wide character for the current locale.
- The macro IFX_GL_MB_MAX is the maximum number of bytes in the multibyte representation for any wide character in any locale.

  The value is always greater than or equal to the value that **ifx_gl_mb_loc_max()** returns.

Of the two options, the macro IFX_GL_MB_MAX is faster and can initialize static buffers. The function **ifx_gl_mb_loc_max()** is slower but more precise.

## Return values

**>=0** The number of bytes written to *mb*.

**-1** The function was not successful, and the error number is set to indicate the cause. See the Errors section.

## Errors

If an error occurred, this function returns -1 and sets the **ifx_gl_lc_errno()** error number to the following value.

**IFX_GL_EILSEQ**
The **wc* value is not a valid wide character. In this case, some bytes might be written to mb, but the contents of *mb* are undefined.

**Related reference**:

"The ifx_gl_lc_errno() function" on page 4-85

"The ifx_gl_mb_loc_max() function" on page 4-86

"The ifx_gl_mbstowcs() function" on page 4-110

"The ifx_gl_mbtowc() function" on page 4-112

"The ifx_gl_wcstombs() function" on page 4-134

# Appendix A. List of Informix GLS error numbers

This topic provides a list of the error-number constants that the **ifx_gl_lc_errno()** function can retrieve. It also provides a list of the IBM Informix GLS functions that might set a particular error number.

The Informix GLS error-number constants are defined in the `gls.h` header file. For a complete list of error-number constants, see the `gls.h` header file.

| Error-number constant | Description | Informix GLS functions |
|---|---|---|
| IFX_GL_NOERRNO | No error has occurred. | Internal use only |

| Error-number constant | Description | Informix GLS functions |
|---|---|---|
| IFX_GL_EILSEQ | Character sequence in a multibyte or wide character is invalid. | **ifx_gl_cv_mconv()**, **ifx_gl_getmb()**, **ifx_gl_ismalnum()**, **ifx_gl_ismalpha()**, **ifx_gl_ismblank()**, **ifx_gl_ismcntrl()**, **ifx_gl_ismdigit()**, **ifx_gl_ismgraph()**, **ifx_gl_ismlower()**, **ifx_gl_ismprint()**, **ifx_gl_ismpunct()**, **ifx_gl_ismspace()**, **ifx_gl_ismupper()**, **ifx_gl_ismxdigit()**, **ifx_gl_iswalnum()**, **ifx_gl_iswalpha()**, **ifx_gl_iswblank()**, **ifx_gl_iswcntrl()**, **ifx_gl_iswdigit()**, **ifx_gl_iswgraph()**, **ifx_gl_iswlower()**, **ifx_gl_iswprint()**, **ifx_gl_iswpunct()**, **ifx_gl_iswspace()**, **ifx_gl_iswupper()**, **ifx_gl_iswxdigit()**, **ifx_gl_mblen()**, **ifx_gl_mbscat()**, **ifx_gl_mbschr()**, **ifx_gl_mbscoll()**, **ifx_gl_mbscpy()**, **ifx_gl_mbscspn()**, **ifx_gl_mbslen()**, **ifx_gl_mbsmbs()**, **ifx_gl_mbsncat()**, **ifx_gl_mbsncpy()**, **ifx_gl_mbsnext()**, **ifx_gl_mbsntsbytes()**, **ifx_gl_mbsntslen()**, **ifx_gl_mbspbrk()**, **ifx_gl_mbsprev()**, **ifx_gl_mbsrchr()**, **ifx_gl_mbsspn()**, **ifx_gl_mbstowcs()**, **ifx_gl_mbtowc()**, **ifx_gl_putmb()**, **ifx_gl_tomlower()**, **ifx_gl_tomupper()**, **ifx_gl_towlower()**, **ifx_gl_towupper()**, **ifx_gl_wcscoll()**, **ifx_gl_wcslen()**, **ifx_gl_wcstombs()**, **ifx_gl_wctomb()** |
| IFX_GL_ENULLPTR | Function received a NULL pointer to a function. | Internal use only |

| Error-number constant | Description | Informix GLS functions |
|---|---|---|
| IFX_GL_ENOMEM | Function encountered a memory-allocation failure. | **ifx_gl_init()**, **ifx_gl_mbscoll()**, **ifx_gl_wcscoll()** |
| IFX_GL_EINDEXRANGE | Index is out of bounds. | Internal use only |
| IFX_GL_EINVPTR | An end pointer is less than a begin pointer. | **ifx_gl_mbsprev()**, **ifx_gl_putmb()** |
| IFX_GL_ERANGE | The base of a number is out of range. | Internal use |
| IFX_GL_EINVAL | Wide-character or multibyte-character string is invalid. | **ifx_gl_conv_needed()**, **ifx_gl_cv_mconv()**, **ifx_gl_getmb()**, **ifx_gl_ismalnum()**, **ifx_gl_ismalpha()**, **ifx_gl_isblank()**, **ifx_gl_ismcntrl()**, **ifx_gl_ismdigit()**, **ifx_gl_ismgraph()**, **ifx_gl_ismlower()**, **ifx_gl_ismprint()**, **ifx_gl_ismpunct()**, **ifx_gl_ismspace()**, **ifx_gl_ismupper()**, **ifx_gl_ismxdigit()**, **ifx_gl_mblen()**, **ifx_gl_mbscat()**, **ifx_gl_mbschr()**, **ifx_gl_mbscoll()**, **ifx_gl_mbscpy()**, **ifx_gl_mbscspn()**, **ifx_gl_mbslen()**, **ifx_gl_mbsmbs()**, **ifx_gl_mbsncat()**, **ifx_gl_mbsncpy()**, **ifx_gl_mbsnext()**, **ifx_gl_mbsntsbytes()**, **ifx_gl_mbsntslen()**, **ifx_gl_mbspbrk()**, **ifx_gl_mbsprev()**, **ifx_gl_mbsrchr()**, **ifx_gl_mbsspn()**, **ifx_gl_mbstowcs()**, **ifx_gl_mbtowc()**, **ifx_gl_putmb()**, **ifx_gl_tomlower()**, **ifx_gl_tomupper()** |
| IFX_GL_FILEERR | Input file could not be read. | **ifx_gl_cv_mconv()**, **ifx_gl_cv_outbuflen()**, **ifx_gl_cv_sb2sb_table()**, **ifx_gl_init()** |

| Error-number constant | Description | Informix GLS functions |
|---|---|---|
| IFX_GL_PARAMERR | Parameter is out of bounds. | **ifx_gl_convert_number()**, **ifx_gl_format_number()**, **ifx_gl_init()**, **ifx_gl_mbscat()**, **ifx_gl_mbschr()**, **ifx_gl_mbscoll()**, **ifx_gl_mbscpy()**, **ifx_gl_mbscspn()**, **ifx_gl_mbslen()**, **ifx_gl_mbsmbs()**, **ifx_gl_mbsncat()**, **ifx_gl_mbsncpy()**, **ifx_gl_mbsntsbytes()**, **ifx_gl_mbsntslen()**, **ifx_gl_mbspbrk()**, **ifx_gl_mbsrchr()**, **ifx_gl_mbsspn()**, **ifx_gl_mbstowcs()**, **ifx_gl_wcscat()**, **ifx_gl_wcschr()**, **ifx_gl_wcscoll()**, **ifx_gl_wcscpy()**, **ifx_gl_wcscspn()**, **ifx_gl_wcsncat()**, **ifx_gl_wcsncpy()**, **ifx_gl_wcsntslen()**, **ifx_gl_wcspbrk()**, **ifx_gl_wcsrchr()**, **ifx_gl_wcsspn()**, **ifx_gl_wcstombs()**, **ifx_gl_wcswcs()** |
| IFX_GL_CATASTROPHE | Function encountered an internal error. Result is undefined result. | Internal use only |
| IFX_GL_BADFILEFORM | File format was invalid. | **ifx_gl_conv_needed()**, **ifx_gl_init()** |
| IFX_GL_INVALIDLOC | Locale code sets are inconsistent. | **ifx_gl_init()** |
| IFX_GL_EIO | Function encountered an I/O error. | Internal use only |
| IFX_GL_E2BIG | Operation would overflow a buffer. | **ifx_gl_convert_date()**, **ifx_gl_cv_mconv()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()**, **ifx_gl_format_money()**, **ifx_gl_format_number()** |
| IFX_GL_EBADF | Function received a bad handle. | **ifx_gl_conv_needed()**, **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_EOF | Function encountered an end-of-file on input stream. | **ifx_gl_getmb()** |
| IFX_GL_EUNKNOWN | An unknown system error has occurred. | Internal use only |

| Error-number constant | Description | Informix GLS functions |
|---|---|---|
| IFX_GL_UNLOADEDCAT | Function cannot copy from an unloaded category. | Internal use only |
| IFX_GL_LOADEDCAT | Function cannot copy into a loaded category. | Internal use only |
| IFX_GL_ENOSYS | A feature is not supported. | **ifx_gl_format_money()** |
| IFX_GL_ELOCTOOWIDE | The current locale has characters that are too wide for this version of the Informix GLS library. | **ifx_gl_init()** |
| IFX_GL_INVALIDFMT | A formatted argument string is invalid. | **ifx_gl_convert_date()**, **ifx_gl_convert_money()**, **ifx_gl_convert_number()**, **ifx_gl_format_number()** |
| IFX_GL_EFRACRANGE | Fraction of second is out of bounds. | **ifx_gl_convert_datetime()**, **ifx_gl_format_datetime()** |
| IFX_GL_ESECONDRANGE | Second is out of bounds. | **ifx_gl_convert_datetime()**, **ifx_gl_format_datetime()** |
| IFX_GL_EMINUTERANGE | Minute is out of bounds. | **ifx_gl_convert_datetime()**, **ifx_gl_format_datetime()** |
| IFX_GL_EHOURRANGE | Hour is out of bounds. | **ifx_gl_convert_datetime()**, **ifx_gl_format_datetime()** |
| IFX_GL_EDAYRANGE | Day number is out of bounds. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_EWKDAYRANGE | Weekday number is out of bounds. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_EYDAYRANGE | Year-day number is out of bounds. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_EMONTHRANGE | Month number is out of bounds. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_EERAOFFRANGE | Era offset is out of bounds. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_EYEARRANGE | Year number is out of bounds. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADFRAC | Fraction could not be scanned. | **ifx_gl_convert_datetime()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADSECOND | Second could not be scanned. | **ifx_gl_convert_datetime()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADMINUTE | Minute could not be scanned. | **ifx_gl_convert_datetime()**, **ifx_gl_format_datetime()** |

| Error-number constant | Description | Informix GLS functions |
|---|---|---|
| IFX_GL_BADHOUR | Hour could not be scanned. | **ifx_gl_convert_datetime()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADDAY | Month-day could not be scanned. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADWKDAY | Weekday could not be scanned. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADYDAY | Year-day could not be scanned. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADMONTH | Month could not be scanned. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADYEAR | Year could not be scanned. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADERANAME | Era name is invalid. | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADERAOFFSET | Era offset is invalid | **ifx_gl_convert_date()**, **ifx_gl_convert_datetime()**, **ifx_gl_format_date()**, **ifx_gl_format_datetime()** |
| IFX_GL_BADFMTMOD | Format modifier is invalid. | **ifx_gl_convert_date()** |
| IFX_GL_BADFMTWP | Field width or precision is invalid. | **ifx_gl_convert_date()** |
| IFX_GL_BADINPUT | Input string does not match format string. | **ifx_gl_convert_date()** |
| IFX_GL_NOPOINT | Input is missing a decimal separator. | **ifx_gl_convert_date()** |
| IFX_GL_BADMONTHSTR | Month string could not be scanned | **ifx_gl_convert_date()** |
| IFX_GL_BADERASPEC | Function could not load era from locale. | **ifx_gl_convert_date()** |
| IFX_GL_BADCALENDAR | LC_TIME category contains an unsupported calendar. | Internal use only |
| IFX_GL_BADOBJVER | The lc (locale), cm (code set), or cv (code-set conversion) object is the wrong version. | **ifx_gl_init()** |
| IFX_GL_BADALTDATE | Function could not convert %Z information. | Internal use only |
| IFX_GL_NOSYMMAP | Character/symbol is not in charmap. | Internal use only |

| Error-number constant | Description | Informix GLS functions |
|---|---|---|
| IFX_GL_BADSYM | Symbolic character name is invalid. | Internal use only |
| IFX_GL_ELOCLOAD | Function could not load locale. | Internal use only |
| IFX_GL_TERMMISMAT | String-termination parameters do not match. | **ifx_gl_mbscat()**, **ifx_gl_mbscoll()**, **ifx_gl_mbscspn()**, **ifx_gl_mbsmbs()**, **ifx_gl_mbsncat()**, **ifx_gl_mbspbrk()**, **ifx_gl_mbsspn()**, **ifx_gl_wcscat()**, **ifx_gl_wcscoll()**, **ifx_gl_wcscspn()**, **ifx_gl_wcsncat()**, **ifx_gl_wcspbrk()**, **ifx_gl_wcsspn()**, **ifx_gl_wcswcs()** |
| IFX_GL_NOCTYPE | GL_CTYPE is not loaded | Internal use only |
| IFX_GL_LOCMISMAT | Loaded code sets are not same. | Internal use only |

**Related concepts**:

"Informix GLS exceptions" on page 1-8

# Appendix B. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

## Accessibility features for IBM Informix products

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

### Accessibility features

The following list includes the major accessibility features in IBM Informix products. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

### Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

### Related accessibility information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software.

### IBM and accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the IBM commitment to accessibility.

## Dotted decimal syntax diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, that element is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 refers to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

**?** Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

**!** Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

***** Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be

repeated. For example, if you hear the line `5.1* data-area`, you know that you can include more than one data area or you can include none. If you hear the lines `3*, 3 HOST`, and `3 STATE`, you know that you can include `HOST`, `STATE`, both together, or nothing.

**Notes:**

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The `*` symbol is equivalent to a loop-back line in a railroad syntax diagram.

`+` Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line `6.1+ data-area`, you must include at least one data area. If you hear the lines `2+`, `2 HOST`, and `2 STATE`, you know that you must include `HOST`, `STATE`, or both. As for the * symbol, you can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the `*` symbol, is equivalent to a loop-back line in a railroad syntax diagram.

# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other 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 grant 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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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 document 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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

# Index

# E



# G



# H



# I

# L



# M



# N



# O



# P



# S

# U



# V



# W



# X

Printed in USA

SC27-3849-01

Spine information:

Informix Product Family Informix Global Language Support **Version 5.00**

**IBM Informix GLS API Programmer's Guide**