INFORMIX®-CLI

Programmer's Manual

INFORMIX-OnLine Dynamic Server, Version 7.2x
INFORMIX-OnLine Workgroup Server, Version 7.2x
INFORMIX-OnLine XPS, Version 8.1x
INFORMIX-SE, Version 7.2x
INFORMIX-Universal Server, Version 9.1x

Version 2.8
October 1997
Part No. 000-4165

Published by INFORMIX® Press

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

Documentation Team: Twila Booth, Lynne Casey, Jennifer Leland

Table of Contents

Introduction

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

About This Manual

This manual is a user guide and reference manual for INFORMIX-CLI, which is the Informix implementation of the Microsoft *Open Database Connectivity* (ODBC) interface, Version 2.5. This manual explains how to use the INFORMIX-CLI application programming interface (API) to access an Informix database and interact with an Informix database server.

Types of Users

This manual is for C programmers who are using INFORMIX-CLI to access Informix relational databases. This manual assumes that you know C programming and are familiar with the structure of relational databases.

If you have limited experience with relational databases, SQL, or your operating system, see the *Getting Started* manual for your database server for a list of supplementary manuals.

Software Dependencies

This manual assumes that you are using INFORMIX-CLI, Version 2.8, on either a Windows NT, Windows 95, or UNIX platform. In addition, you must use one of the following Informix database servers:

- INFORMIX-OnLine Dynamic Server, Version 7.2x
- INFORMIX-OnLine Workgroup Server, Version 7.2x
- INFORMIX-OnLine XPS, Version 8.1x
- INFORMIX-SE, Version 7.2x
- INFORMIX-Universal Server, Version 9.1x

Demonstration Database

The DB-Access utility, which is provided with your Informix database server products, includes a script to build a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. Sample command files are also included. Some database server software allows you to build other demonstration databases as well.

Many examples in Informix manuals are based on the **stores7** demonstration database. For more information about installing **stores7**, see the *DB-Access User Manual* for your database server.

Documentation Conventions

This section describes the following conventions:

- Typographical conventions
- Icon conventions
- Screen-illustration conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
`monospace`	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of feature-, product-, platform-, or compliance-specific information.
→	This symbol indicates a menu item. For example, “Choose **Tools→Options**” means choose the **Options** item from the **Tools** menu.

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

Icon Conventions

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

Comment Icons

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

Icon	Description
⚠	The *warning* icon identifies vital instructions, cautions, or critical information.
⇨	The *important* icon identifies significant information about the feature or operation that is being described.
💡	The *tip* icon identifies additional details or shortcuts for the functionality that is being described.

Feature, Product, and Platform Icons

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

Icon	Description
GLS	Identifies information that is specific to the Informix Global Language Support (GLS) feature.
IUS	Identifies information that is specific to INFORMIX-Universal Server.
ODS	Identifies information that is specific to INFORMIX-OnLine Dynamic Server.
OWS	Identifies information that is specific to INFORMIX-OnLine Workgroup Server.
SE	Identifies information that is specific to INFORMIX-SE.
UNIX	Identifies information that is specific to the UNIX operating system.
Windows	Identifies information that is specific to both Windows NT and Windows 95 environments.
XPS	Identifies information that is specific to INFORMIX-OnLine XPS.

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

Compliance Icons

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

Icon	Description
Core	Identifies that a function supports the Core ODBC API conformance level.
Level 1	Identifies that a function supports the Level 1 ODBC API conformance level.
Level 2	Identifies that a function supports the Level 2 ODBC API conformance level.

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

Windows

Screen-Illustration Conventions

The illustrations in this manual represent a generic rendition of various windowing environments. The details of dialog boxes, controls, and windows were deleted or redesigned to provide this generic look. Therefore, the illustrations in this manual depict the ODBC Administrator graphical interface a little differently than the way it appears on your screen. ♦

Additional Documentation

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

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

On-Line Manuals

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

Printed Manuals

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com. When you place an order, please provide the following information:

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

Windows

On-Line Help

The ODBC Administrator graphical interface contains Help screens that provides information on how to configure a data source. ♦

Error Message Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. The **finderr** utility displays these error messages on the screen. See the Introduction to the *Informix Error Messages* manual for a detailed description of these error messages.

UNIX

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

Windows

Informix Find Error is a graphical tool. This utility has been created with Microsoft help facilities. For more information, see the Introduction to the *Informix Error Messages* manual. ♦

Documentation Notes and Release Notes

In addition to printed documentation, the following on-line files supplement the information in this manual. For UNIX, these files are located in the **$INFORMIXDIR/release/en_us/0333** directory. For Windows, these files are located in the **$INFORMIXDIR\release\en_us\04e4** directory.

On-Line File	Purpose
CLIDOC_2.8	The documentation-notes file describes features that are not covered in this manual or that have been modified since publication. For Windows, click the **INFORMIX-CLI Document Notes** icon.
CLIENTS_2.0	The **CLIENTS_2.0** file lists the release-notes files for the 2.0 Client SDK. These release-notes files describe feature differences from earlier versions of Informix products and how these differences might affect current products. These files also contain information about any known problems and their workarounds. For Windows, click the **INFORMIX-CLI Release Notes** icon.
CLI_2.8	UNIX The machine notes file describes any special actions that are required to configure and use Informix products on your computer. ♦

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

Compliance with Industry Standards

INFORMIX-CLI is based on Version 2.5 of the Microsoft Open Database Connectivity specification, which in turn is based on the X/Open Group SQL Access Call-Level Interface (CLI) specification. The ODBC and CLI specifications provide a common and open interface through which ANSI-compliant SQL is passed.

Informix Welcomes Your Comments

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

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

Write to us at the following address:

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

If you prefer to send email, our address is:

doc@informix.com

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

650-926-6571

We appreciate your feedback.

Chapter

1

Overview of INFORMIX-CLI

This chapter introduces INFORMIX-CLI, describes the conformance, isolation, and lock levels, and discusses the UNIX environment variables.

What Is INFORMIX-CLI?

INFORMIX-CLI is the Informix implementation of the Microsoft Open Database Connectivity (ODBC) standard. INFORMIX-CLI is a Call Level Interface that supports SQL statements with a library of C functions. An application calls these functions to implement ODBC functionality.

Architecture

INFORMIX-CLI consists of the following parts:

- The INFORMIX-CLI libraries

  These libraries provides the functions and values for the INFORMIX-CLI application programming interface (API).
- A driver manager

  A driver manager provides an interface between an INFORMIX-CLI application and the INFORMIX-CLI driver. It also checks parameters and transitions. The following table describes the driver manager requirements and options for each platform.

Environment	Driver Manager
UNIX	On UNIX, a driver manager is optional. You may purchase a UNIX driver manager from a third-party vendor.
Windows	In Windows environments, INFORMIX-CLI provides the driver manager, which is mandatory.

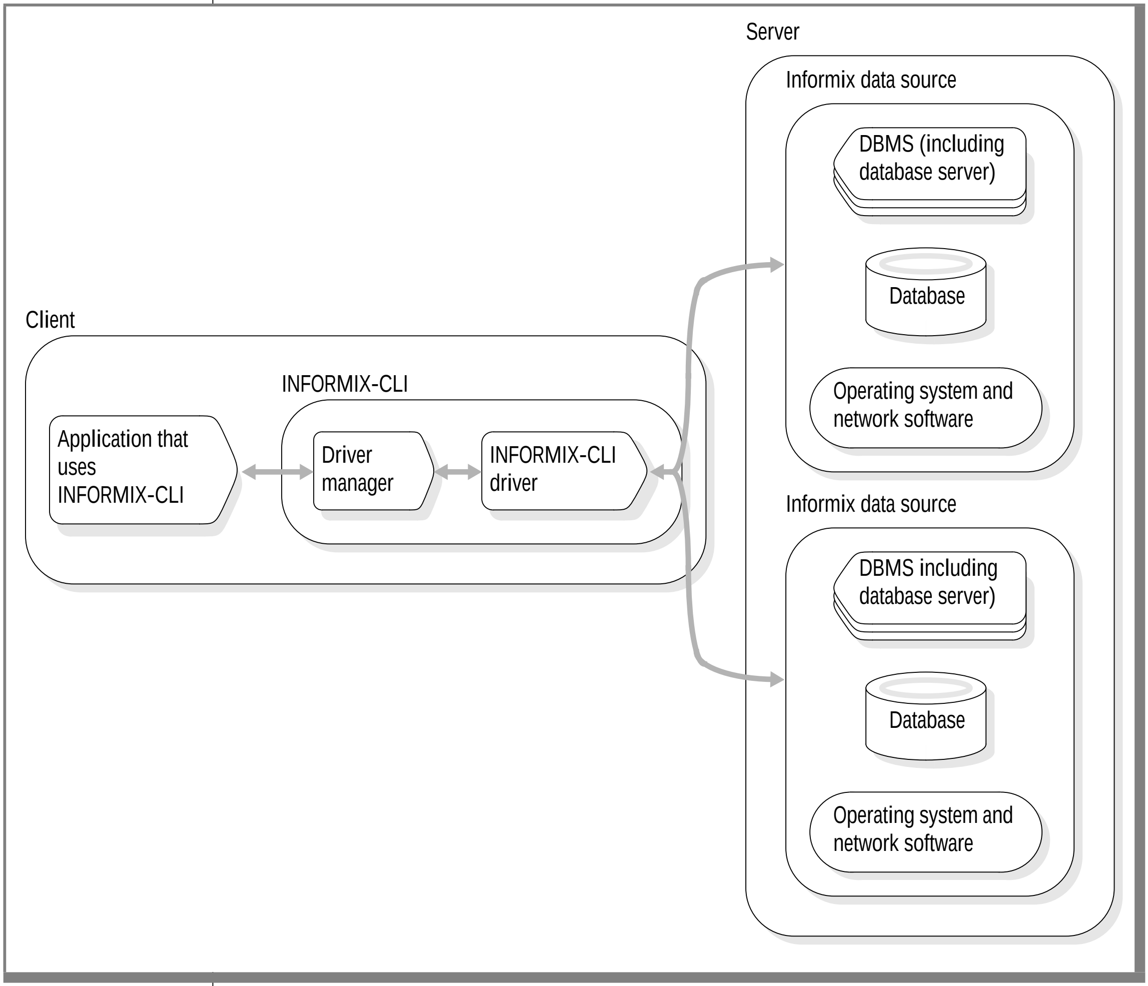
- The INFORMIX-CLI driver

  The driver provides an interface between a data source and either a driver manager or an application, depending on whether or not the system architecture includes a driver manager. If a driver manager is not included, the INFORMIX-CLI driver performs the driver manager functions.

Architecture with a Driver Manager

Figure 1-1 on page 1-5 shows the INFORMIX-CLI architecture when a driver manager is included in the system. In such a system, the driver and driver manager look like a single unit that processes INFORMIX-CLI function calls.

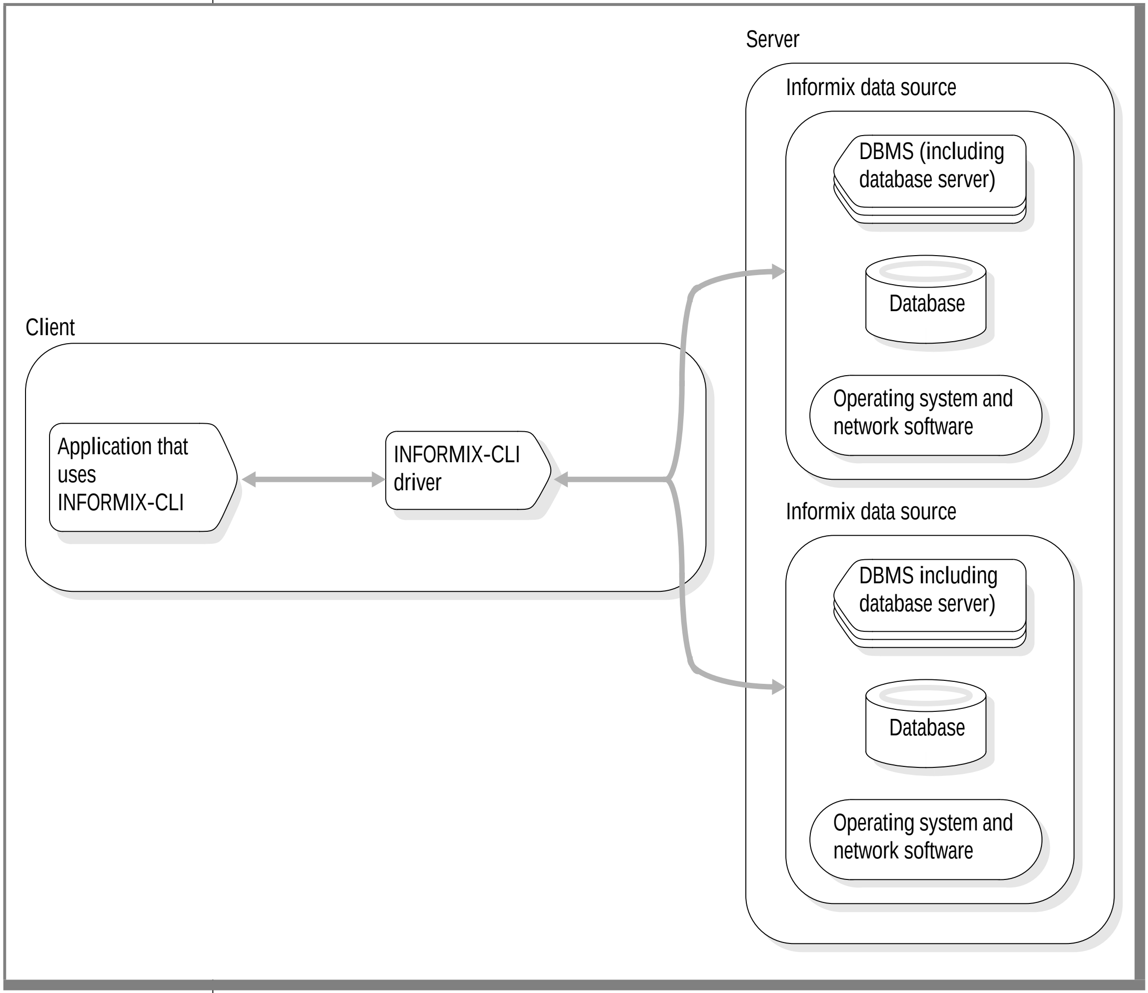
Figure 1-1
INFORMIX-CLI Architecture with a Driver Manager


Server
Informix data source
DBMS (including database server)
Database
Operating system and network software
Informix data source
DBMS including database server)
Database
Operating system and network software
Client
INFORMIX-CLI
Application that uses INFORMIX-CLI
Driver manager
INFORMIX-CLI driver

Architecture Without a Driver Manager

Figure 1-2 shows the INFORMIX-CLI architecture without a driver manager. In this type of system, the application needs to link to the INFORMIX-CLI libraries. See "INFORMIX-CLI Libraries" on page 1-8.

Figure 1-2
INFORMIX-CLI Architecture Without a Driver Manager


Server
Informix data source
DBMS (including database server)
Database
Operating system and network software
Informix data source
DBMS including database server)
Database
Operating system and network software
Client
Application that uses INFORMIX-CLI
INFORMIX-CLI driver

What Is a Data Source?

A data source consists of the following parts:

- A database management system (DBMS), including a database server
- A database
- The operating system and network software necessary for accessing the database

Advantages of Using INFORMIX-CLI

INFORMIX-CLI has the following advantages:

- It lets you develop applications that:
    - connect to and disconnect from data sources.
    - retrieve information about data sources.
    - retrieve information about INFORMIX-CLI.
    - set and retrieve INFORMIX-CLI options.
    - prepare and send SQL statements.
    - retrieve SQL results and process the results dynamically.
    - retrieve information about SQL results and process the information dynamically.
- It is straightforward for application developers who are familiar with function libraries
- It does not use a preprocessor
- It lets you allocate storage for results before or after the results are available. This feature lets you determine the results and the action to take without the limitations that predefined data structures impose.

Isolation and Lock Levels

ODS

IUS

If an INFORMIX-CLI application is connected to INFORMIX-Universal Server or INFORMIX-OnLine Dynamic Server, INFORMIX-CLI supports the following isolation levels:

- 0 (Read Uncommitted)
- 1 (Read Committed)
- 3 (Serializable)

The default is 1. ♦

INFORMIX-CLI also supports an alternative isolation level 1, called cursor stability. For information about this option, see "SQLSetConnectOption" on page 12-281.

INFORMIX-CLI supports transactions only if transaction logging has been enabled for your database. The driver is always in auto-commit mode which treats each statement as a single transaction.

INFORMIX-CLI Libraries

For the locations and filenames of the INFORMIX-CLI libraries, see the Release Notes.

UNIX

INFORMIX-CLI provides two versions of its libraries: shared and static. To link an application with an INFORMIX-CLI library, specify an additional link-library search path for the INFORMIX-CLI library and other library names. For example:

```
cc .... -L$INFORMIXDIR/lib/cli  -libifcli -laio -lm -lnsl -lsocket
```

This command finds the archive library, the shared library, and other system-dependent libraries. ♦

UNIX

Environment Variables on UNIX

Set the **INFORMIXDIR** environment variable to the full path of the directory where your Informix product is installed. The UNIX environment variable, **PATH**, indicates the directories that are searched for executable programs. Your **PATH** setting must include the path to your **$INFORMIXDIR/bin** directory.

Setting Environment Variables in a File

If you set these variables at the command line, you must reset them whenever you log on to your system. If you set these variables in a file, they are set automatically when you log on to your system.

For example, if your Informix directory path is **/usr/informix**, in the C shell you can add the following lines to your **.cshrc** file to set the **INFORMIXDIR** and **PATH** environment variables:

```
setenv INFORMIXDIR /usr/informix
setenv PATH ${PATH}:${INFORMIXDIR}/bin
```

In the Bourne or Korn shells, you can add the following lines to your **.login** or **.profile** file:

```
INFORMIXDIR=/usr/informix; export INFORMIXDIR
PATH=${PATH}:$INFORMIXDIR/bin; export PATH
```

Setup File

INFORMIX-CLI provides a sample setup file called **setup.odbc** in **$INFORMIXDIR/etc.** You can use this file to set up environment variables for the INFORMIX-CLI driver. This file includes the following environment variables:

- **INFORMIXDIR**

  INFORMIXDIR specifies the directory where the INFORMIX-Enterprise Client Software Developer's Kit is installed.

- **INFORMIXSQLHOSTS**

  INFORMIXSQLHOSTS is optional. It specifies the directory that contains **sqlhosts**. By default, **sqlhosts** is in **$INFORMIXDIR/etc.** Set **INFORMIXSQLHOSTS** if you want **sqlhosts** to be in a different directory.

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

GLS

For information about the Global Language Support (GLS) environment variables, see the *Informix Guide to GLS Functionality*. ♦

GLS

GLS

Informix products can support many languages, cultures, and code sets. GLS provides support for all language- and culture-specific information. The following table describes how to set the GLS options depending on your platform..

Environment	How to Set GLS Options
UNIX	Specify the GLS options in the **odbc.ini** file. For more information, see "Configuring Data Sources on UNIX" on page 2-9.
Windows	Specify the GLS options in the INFORMIX-CLI DSN Setup dialog box. For more information, see "Configuring Data Sources on Windows" on page 2-3.

The rest of this section describes the GLS options for INFORMIX-CLI. For more information about GLS and locales, see the *Informix Guide to GLS Functionality*.

Client Locale

Description:	Locale and codeset that the application runs in
Format:	**locale.codeset@modifier**
odbc.ini field for UNIX:	CLIENT_LOCALE
Default value for Windows:	**en_us.1252**

Database Locale

Description:	Locale and codeset that the database was created in
Format:	**locale.codeset@modifier**
odbc.ini field for UNIX:	DB_LOCALE
Default value for Windows:	**en_us.1252**

Translation Library

Description:	Performs the codeset conversion
Format:	Path to the file for the library. The translation DLL must follow the ODBC standard for translation libraries. For more information, see "SQLSetConnectOption" on page 12-281.
odbc.ini field for UNIX:	TRANSLATIONDLL
Default value for Windows:	**%INFORMIXDIR%\bin\igo4n304.dll**

Translation Option

Description:	Option for a non-Informix translation library
Format:	Determined by the vendor
odbc.ini field for UNIX:	TRANSLATION_OPTION
Default value for Windows:	Determined by the vendor

Important: *Do not set this option for an Informix translation library. An Informix translation library determines the translation option based on the client locale and database locale values.*

VMB Character

Description:	Varying multibyte character length reporting option that specifies how to set *pcbValue* when *rgbValue* (the output area) is not large enough for the code-set-converted data. The possible values are: **Estimate** = INFORMIX-CLI makes a worst-case estimate of the storage space needed to return the data. This is the default value. **Exact** = INFORMIX-CLI writes the code-set-converted data to disk until all of the data is converted. Because this option can degrade performance, Informix recommends that you do not use this option unless your application does not work with the **Estimate** option. When you use a multibyte code set (in which characters vary in length, ranging from 1 to 4 bytes) as either the database or client locale, the length of a character string or simple large object (TEXT) in the database locale does not indicate the length of the string after it is converted to the client locale.
Possible values for UNIX:	0 = **Estimate** 1 = **Exact**
Possible values for Windows:	**Estimate** **Exact**
odbc.ini field for UNIX:	VMBCHARLENEXACT
Default value for Windows:	**Estimate**

♦

Chapter

Configuring Data Sources

2

This chapter explains how to configure data sources. Before you can connect to a data source, you must configure it. For information about data sources, see “What Is a Data Source?” on page 1-7.

Windows

Configuring Data Sources on Windows

In Windows environments, INFORMIX-CLI provides the ODBC Administrator, which lets you configure data sources. The ODBC Administrator provides a graphical user interface (GUI) that simplifies the process of adding and modifying data sources. After you add or change data-source configuration information, the ODBC Administrator updates the appropriate files to reflect the specified values.

Warning: *To be compatible with other Informix connectivity products, INFORMIX-CLI no longer stores the data source configuration information in the* ***odbc.ini*** *file. Instead, Informix products store data source configuration information in a set of files. Therefore, modifying* ***odbc.ini*** *has unpredictable system results. Use the ODBC Administrator to add or modify data sources. This warning applies only to Windows environments.*

Adding a Data Source

When you add a data source, you must provide the name of the data source and the name of the database to which you want to connect to by default. All other connection information is optional.

The following table lists the required information that you must set when you add a data source.

Keyword-Attribute	Description
Data Source Name	Name of a data source as returned by **SQLDataSources** or the data sources dialog box of **SQLDriverConnect**
Database	Name of the database to which you want to connect by default

The following table lists optional information that you can set when you add a data source.

Attribute	Description
Host	The name of the computer on which the Informix database server resides
Protocol	The protocol used to communicate with the database server In Windows environments, values can be *xx*SOCTCP or *xx*SOCSPX where *xx* represents OL, ON, or SE.
PWD	The user's password for the database server
Server	The name of the Informix database server on which the database that you want to access resides
Service	The name assigned to the Informix database server process that runs on your UNIX computer Confirm the service name with your system administrator.
UID	The user ID or account name for access to the data source

To add a data source

1. Start the ODBC Administrator.

   The Data Sources dialog box appears, as Figure 2-1 illustrates.

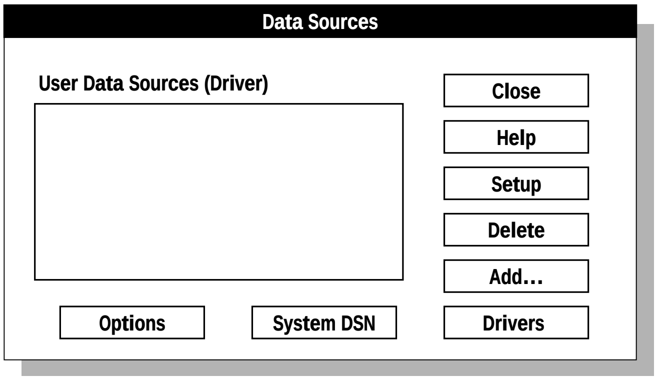
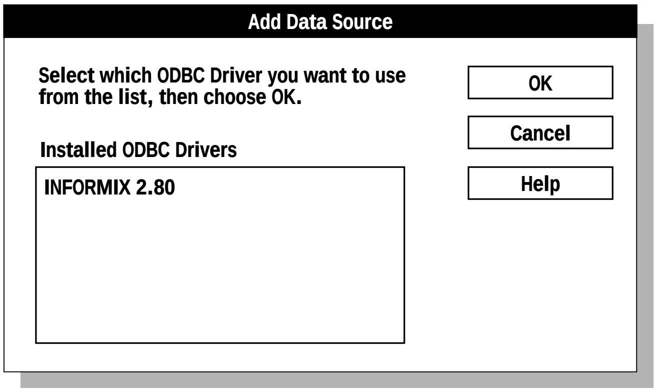


Figure 2-1
Data Sources Dialog Box

2. Click **Add**.

   The Add Data Source dialog box appears, as Figure 2-2 illustrates.



Select which ODBC Driver you want to use from the list, then choose OK.

Installed ODBC Drivers

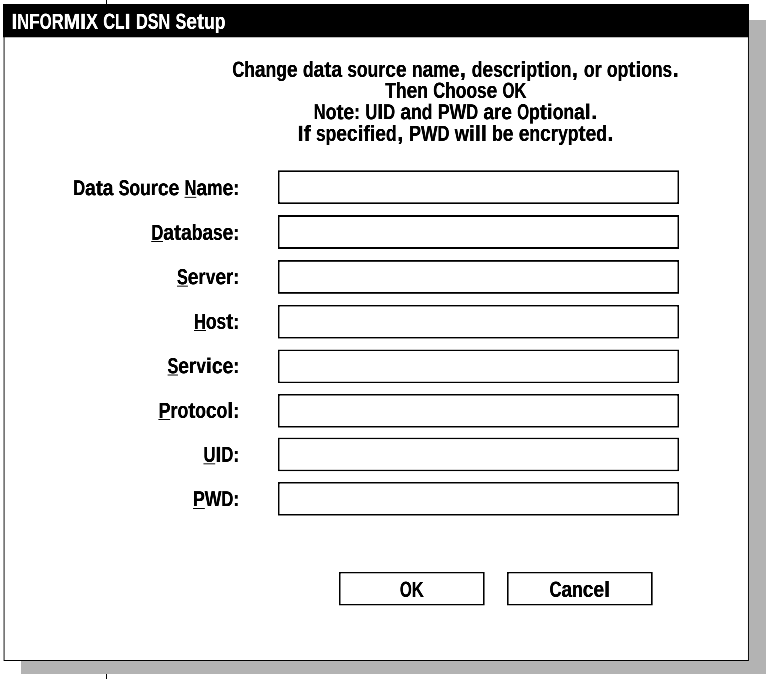
INFORMIX 2.80

Figure 2-2
Add Data Source Dialog Box

3. Select **Informix 2.80** from the Installed ODBC Drivers list.

4. Click **OK**.

   The INFORMIX-CLI DSN Setup dialog box appears, as Figure 2-3 illustrates.



INFORMIX CLI DSN Setup

Change data source name, description, or options.
Then Choose OK
Note: UID and PWD are Optional.
If specified, PWD will be encrypted.

Data Source Name:
Database:
Server:
Host:
Service:
Protocol:
UID:
PWD:

Figure 2-3
INFORMIX-CLI DSN Setup Dialog Box

5. In the **Data Source Name** text box, enter the name of the data source to access.

   You define the **Data Source Name**; that is, it can be any name that you choose. The **Data Source Name** is like an envelope that contains all relevant connection information about the data source.

6. In the **Database** text box, enter the name of the database to which you want to connect by default.

   You now have entered enough information to connect to the data source.

7. Enter any additional information that you want to specify about this data source. For a description of optional data, see "Adding a Data Source" on page 2-4.

8. Click **OK.**

   The Data Sources dialog box appears again.

9. To add another data source, click **Add**. To exit the Data Sources dialog box, click **Close**.

When you click **OK** in the INFORMIX-CLI DSN Setup dialog box, the ODBC Administrator updates the data source information in the appropriate files.

When you use a dialog box or connection string to connect to this data source, the values that you entered are the default entries for the data-source connection.

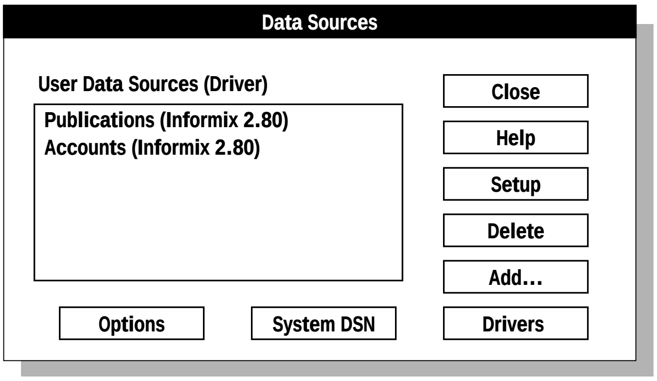
Modifying a Data Source

Data sources that you configured with earlier versions of INFORMIX-CLI are not valid because they specify an earlier version of the INFORMIX-CLI library. To use the features in INFORMIX-CLI, Version 2.8, delete the old data sources and configure new data sources with the INFORMIX-CLI 2.8 driver.

To modify a data source

1. Invoke the ODBC Administrator.

   The Data Sources dialog box appears, as Figure 2-4 illustrates.

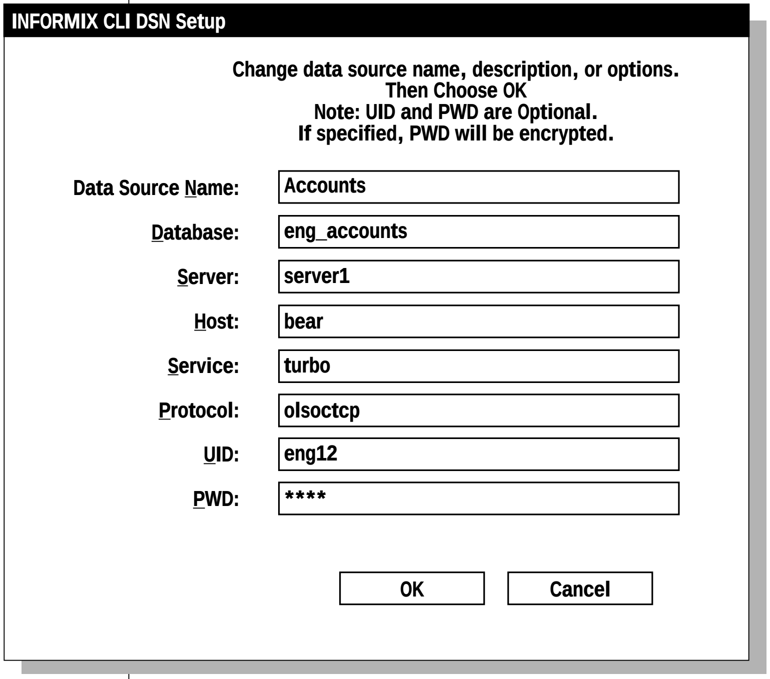


Publications (Informix 2.80)
Accounts (Informix 2.80)

Figure 2-4
Data Sources Dialog Box

2. In the Data Sources dialog box, select the Informix data source that you want to modify and click **Setup**.

   The INFORMIX-CLI DSN Setup dialog box appears, as Figure 2-5 illustrates. The values that appear are the default entries specified for this data-source connection.


INFORMIX CLI DSN Setup

Change data source name, description, or options.
Then Choose OK
Note: UID and PWD are Optional.
If specified, PWD will be encrypted.

Data Source Name:	Accounts
Database:	eng_accounts
Server:	server1
Host:	bear
Service:	turbo
Protocol:	olsoctcp
UID:	eng12
PWD:	****

Figure 2-5
A Completed INFORMIX-CLI DSN Setup Dialog Box

3. Modify the applicable data-source text boxes. For more information regarding available options, see "Adding a Data Source" on page 2-4.
4. When you finish, click **OK**.

   The ODBC Administrator updates the data source information in the appropriate files.

When you connect to this data source using either a dialog box or connection string, the values that you entered appear as the new default entries for the data-source connection. ♦

UNIX

Configuring Data Sources on UNIX

Configuring a data source on UNIX involves the following files:

- **odbc.ini**

  The **odbc.ini** initialization file provides configuration information about data sources. Every data source to which your application connects must have an entry in this file. The information in **odbc.ini** describes the following items:

  - The database to access
  - The server associated with the database
  - The host on which the DBMS resides
  - The network protocol used to access that platform

- **odbcinst.ini**

  The **odbcinst.ini** file provides configuration information about ODBC drivers. You need to modify this file only if you are using an ODBC driver manager from a third-party vendor.

- **sqlhosts**

  The **sqlhosts** file describes the following items:

  - The database server name
  - The type of connection
  - The name of the host computer
  - The service name

  For more information about **sqlhosts**, see the *Administrator's Guide* for your database server.

You can use a text editor to modify these files. The following table explains where the files are located.

File	Description
odbc.ini	When you install INFORMIX-CLI, the installation script installs a sample **odbc.ini** file in **$INFORMIXDIR/etc**. Copy the sample **odbc.ini** file to your HOME directory.
odbcinst.ini	Use the **odbcinst.ini** file that the installation script placed in **$INFORMIXDIR/etc**.
sqlhosts	Use the **sqlhosts** file that the installation script placed in **$INFORMIXDIR/etc**.

File Format for odbc.ini

The following table describes the sections in the **odbc.ini** file.

Section	Description	Status
ODBC Data Sources	This section lists the data sources and associates them with the INFORMIX-CLI driver. You need to provide this section only if you use an ODBC driver manager from a third-party vendor.	Optional
Data Source Specification	Each data source listed in the ODBC Data Sources section has a Data Source Specification section that describes the data source.	Required

ODBC Data Sources

Each entry in the ODBC Data Sources section lists a data source and the INFORMIX-CLI driver name. You need to provide this section only if you use an ODBC driver manager from a third-party vendor.

Format

```
[ODBC Data Sources]
data_source_name = INFORMIX-CLI 2.8 Driver
data_source_name = INFORMIX-CLI 2.8 Driver
.
.
.
```

Field	Description	Status
data_source_name	Data source that an INFORMIX-CLI application can access. Use any data source name that you choose.	Required

Example

The following example defines a data source called EmpInfo:

```
[ODBC Data Sources]
EmpInfo = INFORMIX-CLI 2.8 Driver
```

Data Source Specification

Each data source in the ODBC Data Sources section has a Data Source Specification section.

Format

```
[data_source_name]
Driver = driver_path
Description = data_source_description
Database = database_name
UID = user_id
PWD = user_password
Server = database_server
CLIENT_LOCALE = application_locale
DB_LOCALE = database_locale
TRANSLATIONDLL = translation_path
VMBCHARLENEXACT = output_option
```

Field	Description	Status
data_source_name	Data source specified in the ODBC Data Sources section	Required
driver_path	Path for the INFORMIX-CLI driver	Required
data_source_description	Description of the INFORMIX-CLI driver	Optional
database_name	Name of the database that is associated with the data source	Required
user_id	User ID for connecting to the data source	Optional
user_password	Password for connecting to the data source	Optional
database_server	Name of the database server that is associated with the data source	Required
application_locale	Locale and code set in which the application runs Default value: **en_us.8859-1**	Optional

(1 of 2)

GLS

Field	Description	Status
database_locale	Local and code set in which the database was created Default value: **en_us.8859-1**	Optional
translation_path	Performs the code-set conversion. Default value: **$INFORMIXDIR/lib/esql/ig04a304.xx,** where **xx** represents a platform-specific file extension	Optional
output_option	Varying multibyte character length reporting option that specifies how to set *pcbValue* when *rgbValue* (the output area) is not large enough for the code-set-converted data. Possible values: ■ 0 = Estimate ■ 1 = Exact Default value: 0♦	Optional

(2 of 2)

For more information about the GLS fields, see "GLS" on page 1-11.

Example

The following example shows the configuration for a data source called EmpInfo:

```
[EmpInfo]
Driver = /informix/lib/cli/iclis09a.so
Description = Demo data source
Database = stores7
UID = admin
PWD = tiger
Server = ifmx_91
CLIENT_LOCALE = en_us.8859-1
DB_LOCALE = de_de.646de
TRANSLATIONDLL = /opt/informix/lib/esql/igo4a304.so
VMBCHARLENEXACT = 0
```

File Format for odbcinst.ini

The following table describes the sections in the **odbcinst.ini** file.

Section	Description	Status
ODBC Drivers	Lists the INFORMIX-CLI driver.	Required
Driver Specification	Describes the INFORMIX-CLI driver.	Required
Default Driver Specification	Specifies the driver to use when none is specified. The default driver is the INFORMIX-CLI driver.	Required
ODBC Translators	Lists each available translator.	Required
Translator Specification	Each translator listed in the ODBC Translators section has a Translator Specification section that describes the translator.	Required

Adding a Data Source

1. Add a Data Source Specification section for the data source to **odbc.ini**.
2. If necessary, add an entry for the database server to **sqlhosts**.
3. If you use an ODBC driver manager from a third-party vendor:
    a. If the ODBC Data Sources section does not already exist in **odbc.ini**, create it.
    b. Add an entry for the data source to the ODBC Data Sources section in **odbc.ini**.
    c. If **odbcinst.ini** does not already have an entry for the INFORMIX-CLI driver in the ODBC Drivers section, add the entry.
    d. If **odbcinst.ini** does not already have a Driver Specification section for the INFORMIX-CLI driver, add the section.

Modifying a Data Source

To modify a data source, modify the Data Source Specification section for the data source in **odbc.ini**. ♦

Chapter

3 Guidelines for Calling INFORMIX-CLI Functions

This chapter describes the conformance levels, the driver manager, the general characteristics of INFORMIX-CLI functions, and the basic procedure for using the API.

Conformance Levels

This section describes the API and SQL conformance levels for the INFORMIX-CLI driver.

API Conformance Levels

API conformance refers to the functions that an ODBC driver supports. The API conformance standard consists of three levels: Core, Level 1, and Level 2. The Core level consists of functions that correspond to the functions in the CLI specification. Level 1 and Level 2 functions extend the core functionality.

The following table lists the functions that the ODBC standard defines. INFORMIX-CLI supports all the Core functions and Level 1 functions. INFORMIX-CLI supports the Level 2 functions except those marked with an asterisk (*).

Core Level Functions	Level 1 Functions	Level 2 Functions
SQLAllocConnect	**SQLBindParameter**	**SQLBrowseConnect**
SQLAllocEnv	**SQLColumns**	**SQLColumnPrivileges**
SQLAllocStmt	**SQLDriverConnect**	**SQLDataSources**
SQLBindCol	**SQLGetConnectOption**	**SQLDescribeParam** *
SQLCancel	**SQLGetData**	**SQLDrivers**
SQLColAttributes	**SQLGetFunctions**	**SQLExtendedFetch**
SQLConnect	**SQLGetInfo**	**SQLForeignKeys**
SQLDescribeCol	**SQLGetStmtOption**	**SQLMoreResults**
SQLDisconnect	**SQLGetTypeInfo**	**SQLNativeSql**
SQLError	**SQLParamData**	**SQLNumParams**
SQLExecDirect	**SQLPutData**	**SQLParamOptions** *
SQLExecute	**SQLSetConnectOption**	**SQLPrimaryKeys**
SQLFetch	**SQLSetStmtOption**	**SQLProcedureColumns**
SQLFreeConnect	**SQLSpecialColumns**	**SQLProcedures**
SQLFreeEnv	**SQLStatistics**	**SQLSetPos** *
SQLFreeStmt	**SQLTables**	**SQLSetScrollOptions**
SQLGetCursorName		**SQLTablePrivileges**
SQLNumResultCols		
SQLPrepare		
SQLRowCount		
SQLSetCursorName		
SQLTransact		

SQL Conformance Levels

SQL conformance refers to the SQL grammar that an ODBC driver supports. The SQL conformance standard consists of three levels: Minimum, Core, and Extended. INFORMIX-CLI supports all Minimum and Core SQL grammar, which includes the following statements, expressions, and data types:

- Data Definition Language (DDL) statements:
  - CREATE TABLE
  - DROP TABLE
  - ALTER TABLE
  - CREATE INDEX
  - DROP INDEX
  - CREATE VIEW
  - DROP VIEW
  - GRANT
  - REVOKE
- Data Manipulation Language (DML) statements:
  - Full SELECT
  - INSERT
  - UPDATE
  - SEARCHED
  - DELETE SEARCHED
  - Positioned UPDATE
  - Positioned DELETE
  - Outer joins
  - Unions
- Expressions:
  - Simple (such as A>B+C)
  - Subquery
  - Set functions such as SUM and MIN
- Data types:

  See Appendix B.

INFORMIX-CLI supports the following extended SQL grammar:

- Numeric functions:

abs	**acos**	**asin**	**atan**
atan2	**cos**	**cot**	**exp**
log	**log10**	**mod**	**power**
round	**sin**	**sqrt**	**tan**
truncate			

- Date functions:

curdate	**dayofweek**	**now**
dayofmonth	**month**	**year**

- String functions:

concat	**LTRIM**
length	**RTRIM**

- System function **sysusername**
- Procedure calls
- Data types:

  See Appendix B.

For more information about the numeric, date, string, and system functions, see the *Informix Guide to SQL: Syntax*.

Verifying INFORMIX-CLI Conformance Levels

To verify the API conformance level, call **SQLGetInfo** with the SQL_ODBC_SAG_CLI_CONFORMANCE and SQL_ODBC_API_CONFORMANCE flags.

To verify the SQL conformance level, call **SQLGetInfo** with the SQL_ODBC_SQL_CONFORMANCE flag. To verify SQL conformance for a specific SQL extension, call **SQLGetInfo** with a flag for that extension. To verify SQL conformance for a specific SQL data type, call **SQLGetTypeInfo**.

Using the Driver Manager

A driver manager is a shared library that provides an interface between an INFORMIX-CLI application and the INFORMIX-CLI driver. It also checks parameters and transitions. The following table describes the driver manager requirements and options for each platform.

Environment	Driver Manager
UNIX	On UNIX, a driver manager is optional. You may purchase a UNIX driver manager from a third-party vendor.
Windows	In Windows environments, INFORMIX-CLI provides the driver manager, which is mandatory. The driver manager file is **odbc32.dll**.

The following table describes the actions that the driver manager performs for each function.

Functions	Action
SQLDataSources **SQLDrivers**	The driver manager processes the call. It does not pass the call to the driver.
SQLGetFunctions	The driver manager passes the call to the driver associated with the connection.
SQLAllocEnv **SQLAllocConnect** **SQLSetConnectOption** **SQLFreeConnect** **SQLFreeEnv**	The driver manager calls **SQLAllocEnv**, **SQLAllocConnect**, and **SQLSetConnectOption** in the driver when the application calls a function to connect to the data source (**SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**). The driver manager calls **SQLFreeConnect** and **SQLFreeEnv** in the driver when the application calls **SQLDisconnect**.
SQLConnect **SQLDriverConnect** **SQLBrowseConnect** **SQLError**	The driver manager performs initial processing and then passes the call to the driver that is associated with the connection.
All other functions	The driver manager passes the call to the driver.

If requested, the driver manager records each called function in a trace file. It records the name of the function, the values of the input arguments, and the names of the output arguments as listed in the function definitions.

Tip: *For improved performance, your application can bypass the driver manager and link directly to the driver. This solution reduces computational overhead. If your application takes advantage of this solution, however, it cannot use multiple drivers.*

Calling Functions

Every INFORMIX-CLI function name starts with the prefix **SQL**. Each function accepts one or more arguments. Arguments are defined as input (to the driver) or output (from the driver).

An INFORMIX-CLI application must include the **sql.h** and **sqlext.h** header files. These files define INFORMIX-CLI constants and types and provide function prototypes for the INFORMIX-CLI functions.

Buffers

An application passes data to the driver in an input buffer. The driver returns data to the application in an output buffer. The application must allocate memory for both input and output buffers. If the application uses the buffer to retrieve string data, the buffer must contain space for the null termination byte.

Some functions accept pointers to buffers that are used later by other functions. The application must ensure that these pointers remain valid until all applicable functions have used them. For example, the argument *rgbValue* in **SQLBindCol** points to an output buffer where **SQLFetch** returns the data for a column.

Input Buffers

An application passes the address and length of an input buffer to the driver. The length of the buffer must be one of the following values:

- A length greater than or equal to zero

  This value is the actual length of the data in the input buffer. For character data, a length of zero indicates that the data is an empty (zero length) string. A length of zero is different from a null pointer. If the application specifies the length of character data, the character data does not need to be null-terminated.

- SQL_NTS

  This value specifies that a character data value is null-terminated.

- SQL_NULL_DATA

  This value tells the driver to ignore the value in the input buffer and use a NULL data value instead. It is valid only when the input buffer provides the value of a parameter in an SQL statement.

For character data that contains embedded null characters, the operation of INFORMIX-CLI functions is undefined; for maximum interoperability, it is better not to use them. Informix database servers treat null characters as end-of-string markers or as indicators that no more data exists.

Unless it is specifically prohibited in a function description, the address of an input buffer can be a null pointer. In such cases, the value of the corresponding buffer-length argument is ignored. For more information about input buffers, see "Converting Data from SQL to C" on page B-19.

Output Buffers

An application passes the following arguments to the driver so that the driver can return data in an output buffer:

- The address of the output buffer, to which the driver returns the data

  Unless it is specifically prohibited in a function description, the address of an output buffer can be a null pointer. In such cases, the driver does not return anything in the buffer and, in the absence of other errors, returns SQL_SUCCESS.

  If necessary, the driver converts data before returning it. The driver always null-terminates character data before returning it.
- The length of the buffer

  The driver ignores this value if the returned data has a fixed length in C, as with an integer, real number, or date structure.
- The address of a variable in which the driver returns the length of the data (the length buffer)

  The returned length of the data is SQL_NULL_DATA if the data is a NULL value in a result set. Otherwise, the returned length of the data is the number of bytes of data that are available to return. If the driver converts the data, the returned length of the data is the number of bytes that remain after the conversion; for character data, it does not include the null-termination byte that the driver added.

If the output buffer is too small, the driver attempts to truncate the data. If the truncation does not cause a loss of significant data, the driver returns the truncated data in the output buffer, returns the length of the available data (as opposed to the length of the truncated data) in the length buffer, and returns SQL_SUCCESS_WITH_INFO. If the truncation causes a loss of significant data, the driver leaves the output and length buffers untouched and returns SQL_ERROR. The application calls **SQLError** to retrieve information about the truncation or the error.

For more information about output buffers, see "Converting Data from SQL to C" on page B-19.

Environment, Connection, and Statement Handles

When an application requests them to, the driver and the driver manager allocate storage for information about the environment, each connection, and each SQL statement. The handles to these storage areas are returned to the application, which uses one or more handles in each call to a function.

The INFORMIX-CLI API uses the following types of handles:

- *Environment handles* identify memory storage for global information, including the valid connection handles and the current active connection handle. The environment handle is an HENV variable type. An application uses one environment handle. It must request this handle before it connects to a data source.
- *Connection handles* identify memory storage for information about particular connections. A connection handle is an HDBC variable type. An application must request a connection handle before it connects to a data source. Each connection handle is associated with the environment handle. However, the environment handle can be associated with multiple connection handles.
- *Statement handles* identify memory storage for information about SQL statements. A statement handle is an HSTMT variable type. An application must request a statement handle before it submits SQL requests. Each statement handle is associated with exactly one connection handle. However, each connection handle can be associated with multiple statement handles.

For more information about requesting a connection handle, see "Connecting to a Data Source" on page 4-4. For more information about requesting a statement handle, see "Executing an SQL Statement" on page 5-5.

Using Data Types

Data stored on a data source has an SQL data type. The INFORMIX-CLI driver maps Informix-specific SQL data types to ODBC SQL data types, which are defined in the ODBC SQL grammar. (The driver returns these mappings through **SQLGetTypeInfo**. It also uses the ODBC SQL data types to describe the data types of columns and parameters in **SQLColAttributes** and **SQLDescribeCol**.)

Each SQL data type corresponds to an ODBC C data type. By default, the driver assumes that the C data type of a storage location corresponds to the SQL data type of the column or parameter to which the location is bound. If the C data type of a storage location is not the *default* C data type, the application can specify the correct C data type with the *fCType* argument in **SQLBindCol**, **SQLGetData**, or **SQLBindParameter**. Before the driver returns data from the data source, it converts the data to the specified C data type. Before the driver sends data to the data source, it converts the data from the specified C data type.

For more information about data types, see Appendix B. The C data types are defined in the **sql.h** and **sqlext.h** header files.

Function Return Codes

When an INFORMIX-CLI application calls a function, the driver executes the function and returns a predefined code. The following return codes indicate success, warning, or failure status:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING
- SQL_NEED_DATA

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, the application can call **SQLError** to retrieve additional information about the error. For a complete description of return codes and error handling, see Chapter 7, "Retrieving Status and Error Information."

How an Application Uses the INFORMIX-CLI API

An application uses the INFORMIX-CLI API to make a connection to a data source, issue SQL statements to a data source, process result data dynamically, and terminate a connection.

1. Connect to the data source. While connecting, specify the data-source name and any additional information needed to complete the connection.
2. Process one or more SQL statements:
   a. Place the SQL text string in a buffer. If the statement includes parameter markers, set the parameter values.
   b. If the statement returns a result set, either assign a cursor name for the statement or let the driver assign one.
   c. Submit the statement for prepared or immediate execution.
   d. If the statement creates a result set, you can inquire about the attributes of the result set, such as the number of columns and the name and type of a specific column. For each column in the result set, assign storage and fetch the results.
   e. If the statement causes an error, retrieve error information from the driver and take the appropriate action.
3. End each transaction by committing it or rolling it back.
4. Terminate the connection when it finishes interacting with the data source.

Figure 3-1 shows the function calls that a basic application makes. Depending on your needs, your application can call other functions.

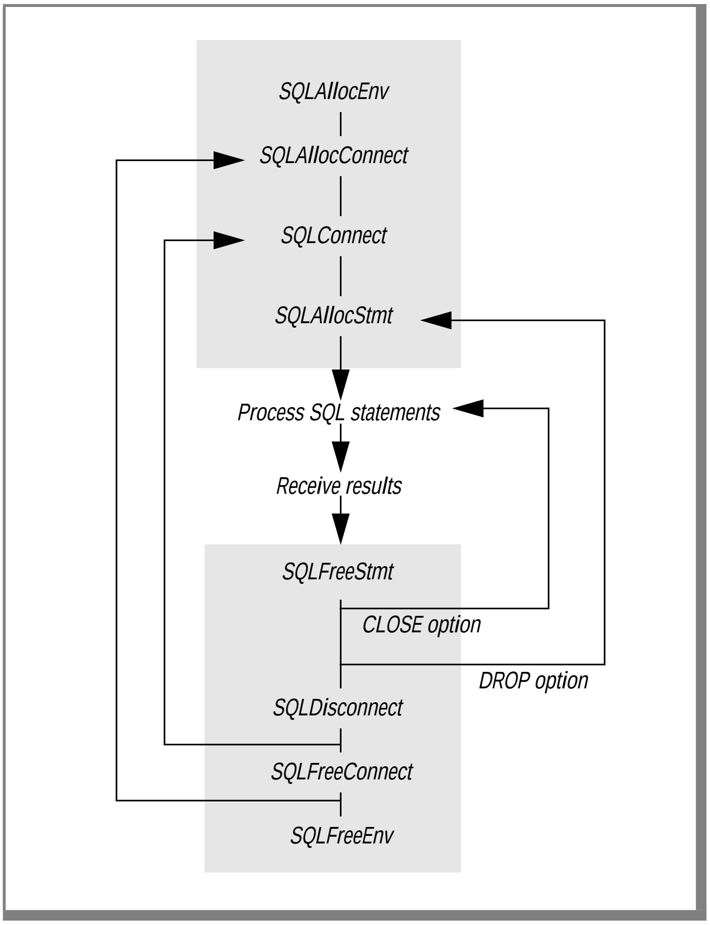


Figure 3-1
Sample Listing of Function Calls That an INFORMIX-CLI Application Makes

Chapter

4 Connecting to a Data Source

This chapter describes the different ways that an application can establish a connection to a data source.

Initializing the Environment

Before an application can use any other function, it must initialize the INFORMIX-CLI API and associate an environment handle with the environment.

To initialize the API and allocate an environment handle

1. Declare a variable of type HENV.

   For example, you could use the following declaration:

   ```
   HENV henv1;
   ```

2. Call **SQLAllocEnv** and pass it the address of the variable.

   The driver initializes the environment, allocates memory to store information about the environment, and returns the environment handle in the variable.

SQLAllocEnv supports one or more connections to data sources; however, your application should perform these steps only once.

Allocating a Connection Handle

Before your INFORMIX-CLI application can connect to the driver, it must allocate a connection handle for the connection.

To allocate a connection handle

1. Declare a variable of type HDBC.

   For example, you could use the following declaration:

   ```
   HDBC hdbc1;
   ```

2. Call **SQLAllocConnect** and pass it the address of the variable.

   The driver allocates memory to store information about the connection and returns the connection handle in the variable.

Connecting to a Data Source

After the application allocates a connection handle, it must specify a data source name or the appropriate connection information. INFORMIX-CLI provides functions that let you use different methods for connecting to a data source.

Using SQLConnect

Your INFORMIX-CLI application passes the following information to the driver in a call to **SQLConnect**:

- The data-source name is the name of the data source that the application is requesting.
- The user ID is the login ID or account name for access to the data source. This value is optional if it was specified in the configuration information. Otherwise, this value is required.
- The password is a character string associated with the user ID that lets the user access the data source. This value is optional if it was specified in the configuration information. Otherwise, this value is required.

If the application specifies a data-source name that was not configured, or if the application does not specify a data-source name, INFORMIX-CLI searches for the default data-source specification. If it finds the default data source, INFORMIX-CLI loads the default driver shared library and passes the application-specified data-source name to that library. If INFORMIX-CLI cannot find a default data source, it returns an error.

Using SQLDriverConnect

Your application can connect to a data source by calling **SQLDriverConnect.**

SQLDriverConnect supports the following features:

- Data sources that require more connection information than the three arguments in **SQLConnect**
- Dialog boxes to prompt the user for all connection information
- Data sources that have not been configured

An application calls **SQLDriverConnect** in one of the following ways:

- Specifies a connection string that contains a data-source name

  INFORMIX-CLI retrieves the full path of the driver shared library associated with the data source. To retrieve a list of data-source names, an application calls **SQLDataSources**.
- Specifies a connection string that contains a driver description

  INFORMIX-CLI retrieves the full path of the driver shared library. To retrieve a list of driver descriptions, an application calls **SQLDrivers**.
- Specifies a connection string that does not contain a data-source name or a driver description

  INFORMIX-CLI displays a dialog box from which the user selects a data-source name. INFORMIX-CLI then retrieves the full path of the driver shared library associated with the data source.

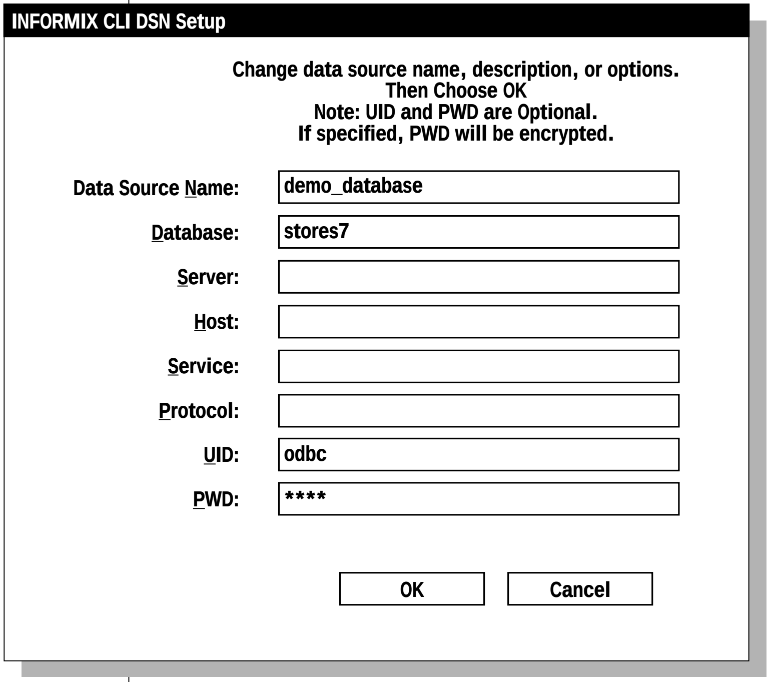
INFORMIX-CLI then loads the driver shared library and passes the **SQLDriverConnect** arguments to it.

The application can pass all the connection information that the driver needs. It can also request that the driver always prompt the user for connection information or only prompt the user for information it needs. Finally, if a data source is specified, the driver can read connection information from the appropriate section of the registry.

After the driver connects to the data source, it returns the connection information to the application. The application can store this information for future use.

If the application specifies a data-source name that was not configured, INFORMIX-CLI searches for the default data-source specification. If it finds the default data source, it loads the default driver shared library and passes the application-specified data-source name to it. If no default data source exists, INFORMIX-CLI returns an error.

When the application calls **SQLDriverConnect** and requests that the user be prompted for information, INFORMIX-CLI displays the INFORMIX-CLI DSN Setup dialog box, as Figure 4-1 illustrates.


INFORMIX CLI DSN Setup

Change data source name, description, or options.
Then Choose OK
Note: UID and PWD are Optional.
If specified, PWD will be encrypted.

Data Source Name:	demo_database
Database:	stores7
Server:	
Host:	
Service:	
Protocol:	
UID:	odbc
PWD:	****

Figure 4-1
INFORMIX-CLI DSN Setup Dialog Box

Using SQLBrowseConnect

SQLBrowseConnect supports an iterative method of listing and specifying the attributes and attribute values that are required to connect to a data source. For each level of a connection, an application calls **SQLBrowseConnect** and specifies the connection attributes and attribute values for that level. First-level connection attributes always include the data-source name or driver description; the connection attributes for later levels are data-source dependent but might include the host, user name, and database.

Whenever an application calls **SQLBrowseConnect**, it validates the current attributes, returns the next level of attributes, and returns a user-friendly name for each attribute. It can also return a list of valid values for those attributes. After an application has specified each level of attributes and values, **SQLBrowseConnect** connects to the data source and returns a complete connection string, which **SQLDriverConnect** can use to connect to the data source at a later time.

Connection Strings

Connection strings are used with both the **SQLDriverConnect** and the **SQLBrowseConnect** functions.

With these functions, a connection string contains the following information:

- Data-source name or driver description
- Zero or one user ID
- Zero or one password
- Zero or more data-source-specific parameter values

The connection string is a more flexible API than the data source name, user ID, and password that **SQLConnect** uses. The application can use the connection string for multiple levels of login authorization or to convey other data-source-specific connection information.

With these functions, a connection string has the following syntax:

```
connection-string ::= empty-string[;] | attribute[;] |
attribute; connection-string
empty-string ::=
attribute ::= attribute-keyword=attribute-value |
DRIVER={attribute-value}
(The braces ({}) are literal; the application must specify
them.)
attribute-keyword ::= DSN | UID | PWD| CONNECTDATABASE |
EXCLUSIVE| HOST| PROTOCOL| SERVER| SERVICE
attribute-value ::= character-string
```

In this example, *character-string* has zero or more characters; *identifier* has one or more characters; *attribute-keyword* is case insensitive; *attribute-value* might be case sensitive; and the value of the DSN keyword does not consist solely of blanks. Because of connection-string and initialization-file grammar, avoid keywords and attribute values that contain the characters [ ]{ }( ),;?*=!@\ . Because of the registry grammar, keywords and data-source names cannot contain the backslash (\).

If any keywords are repeated in the connection string, the driver uses the value associated with the first occurrence of the keyword. If the DSN and DRIVER keywords are included in the same connection string, INFORMIX-CLI uses the keyword that appears first.

The following table describes the attribute-keywords available in INFORMIX-CLI. These attributes are available with both **SQLDriverConnect** and **SQLBrowseConnect**.

Attribute Keyword	Attribute Value Description
DSN	Name of a data source as returned by **SQLDataSources** or the data sources dialog box of **SQLDriverConnect**
DRIVER	Description of the driver as returned by the **SQLDrivers** function
UID	The user ID or account name for access to the data source
PWD	The password that corresponds to the user ID, or an empty string if no password exists for the user ID (PWD=;)

(1 of 2)

Attribute Keyword	Attribute Value Description
ConnectDatabase *	A character string This option is available inside a connection string only. `Yes`= Connect to the database that is specified in the DSN setup or in the configuration string. This is the default setting. `No`= Connect to the database server only.
Database	Name of the database to which you want to connect by default
Exclusive *	A character string This option is available inside a connection string only. `Yes`= Specify that the application has exclusive use of the database. `No`= Share database access. This is the default setting.
Host	The name of the computer on which the Informix database server resides
Protocol	The protocol used to communicate with the database server **Windows** In Windows environments, values can be *xx*SOCTCP or *xx*SOCSPX where *xx* represents OL, ON, or SE. ♦
Server	The name of the database server on which the database that you want to access resides
Service	**UNIX** The name assigned to the database-server process that runs on your UNIX computer Confirm the service name with your system administrator. ♦

* INFORMIX-CLI provides two enhanced connection options with the **SQLDriverConnect** and **SQLBrowseConnect** functions. These options let your application perform the following special connections:

- Exclusive database-use connection
- Server-only connection

These options are available only inside a connection string.

(2 of 2)

Exclusive Database-Use Connection

An exclusive database-use connection allows the application to lock out all other users and applications. When this parameter is set to Yes, the user or application has exclusive use of the database. When it is set to No, other users and applications can access the database.

Server-Only Connection

A server-only connection lets you open a connection between the driver and the database server without connecting to a database. This option lets you create new databases or drop existing databases within an application. However, when you use this option, you can execute only the CREATE DATABASE and DROP DATABASE SQL statements.

To use the server-only connection in an application

1. Enter a connection string that specifies the database server where you want to create or drop a database and set `CONNECTDATABASE` to `NO`.
2. Allocate a statement handle (`Hstmt`).
3. Execute **SQLExecDirect** or **SQLPrepare** and **SQLExecute** with the desired CREATE DATABASE or DROP DATABASE SQL statements.

   For more information on these SQL statements, see the *Informix Guide to SQL: Syntax*.
4. Disconnect from the database server-only connection.
5. Reconnect to the desired database as a normal connection (that is, with the default setting of the ConnectDatabase option).

   Either the configuration information or the connection string must specify the database to which you are connecting.

Example

The following code shows how to use the server-only connection:

```
/*
**      createdb.c
**
**  INFORMIX-CLI CREATE Database Example
**
**
**    OBDC Functions:
**      SQLAllocEnv
**      SQLAllocConnect
**      SQLAllocStmt
**      SQLConnect
**      SQLFreeEnv
**      SQLFreeect
**      SQLFreeStmt
**      SQLDisconnect
**      SQLExecDirect
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <io.h>
#include <windows.h>
#include <conio.h>

#include "infxcli.h"

UCHAR     defDbName[] = "imydsn";

#define ErrMsgLen 200
UCHAR     ErrMsg[ErrMsgLen];


void printError(
    HENV        henv,
    HDBC        hdbc,
    HSTMT       hstmt,
    char       *SqlState )
{
    RETCODE     rc = SQL_NO_DATA_FOUND;
    SWORD       pcbErrorMsgLen = 0;
    SDWORD      pfNativeError;
    short       i;
    for( i = 1; i <= 3 && rc == SQL_NO_DATA_FOUND || rc ==
SQL_INVALID_HANDLE; i++ )
    {
        switch( i )
        {
            case 1:
               rc = SQLError( NULL, NULL, hstmt, (UCHAR
*)SqlState, &pfNativeError, (UCHAR *)ErrMsg,
                              ErrMsgLen, &pcbErrorMsgLen );
               break;

            case 2:
               rc = SQLError( NULL, hdbc, NULL, (UCHAR
*)SqlState, &pfNativeError, (UCHAR *)ErrMsg,
                              ErrMsgLen, &pcbErrorMsgLen );
               break;

            case 3:
               rc = SQLError( henv, NULL, NULL, (UCHAR
*)SqlState, &pfNativeError, (UCHAR *)ErrMsg,
                              ErrMsgLen, &pcbErrorMsgLen );
               break;
        }
    }

    if( rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO )
    {
        fprintf( stdout, "ERROR: %d : %s : %s \r\n",
pfNativeError, SqlState, ErrMsg );
    }
    else
    {
       fprintf( stdout, "ERROR: Severe Error in ODBC Driver
" );
    }
}


int main (
    long         argc,
    char        *argv[] )
{
   // UCHAR        db[20];
    RETCODE      rc = 0;
    HENV         henv = NULL;
    HDBC         hdbc = NULL;
    HSTMT        hstmt = NULL;
    char         SqlState[6];
    UCHAR        ConnStrIn[ 250 ];
    UCHAR       *ConnStrInp = &ConnStrIn[ 0 ];
    UCHAR        ConnStrOut[ 250 ];
    UCHAR       *ConnStrOutp = &ConnStrOut[ 0 ];
    UCHAR       CmdText[] = "create database test1 with log";
    UCHAR       *CmdTextp = &CmdText[ 0 ];
    short        pcbConnStrOut;
    int          in;

    fprintf( stdout, "\n" );

    /* Allocate the Environment handle */
    rc = SQLAllocEnv( &henv );
    if( rc )
    {
        /* note: SQLError cannot be used unless SQLAllocEnv
succeds */
        fprintf( stdout, "Environment Allocation failed\n" );
        goto Exit;
    }

    /* Allocate the Connection handle */
    rc = SQLAllocConnect( henv, &hdbc );
    if( rc )
    {
        /* note: use SQLError from here on */
        printError( henv, hdbc, hstmt, SqlState );
        goto Exit;
    }

    sprintf( ConnStrInp, "dsn=%s;connectdatabase=no",
defDbName ) ;

    /* Establish the server connection */
    rc = SQLDriverConnect( hdbc, NULL, ConnStrInp, SQL_NTS,
                           ConnStrOutp, 250, &pcbConnStrOut,
                           SQL_DRIVER_NOPROMPT );
    if( rc == SQL_ERROR )
    {
        printError( henv, hdbc, hstmt, SqlState );
        goto Exit;
    }

    /* Allocate the statement handle */
    rc = SQLAllocStmt( hdbc, &hstmt );
    if( rc == SQL_ERROR )
    {
        printError( henv, hdbc, hstmt, SqlState );
        goto Exit;
    }
    fprintf( stdout, "Creating database ... \n" );

    /* Create database stores20 */
    rc = SQLExecDirect( hstmt, CmdTextp, SQL_NTS );
    if( rc == SQL_ERROR )
    {
        printError( henv, hdbc, hstmt, SqlState );
        goto Exit;
    }

    fprintf( stdout, "Database successfully created!!!\n" );

    Exit:
       SQLFreeConnect( hdbc );
       SQLFreeEnv( henv );

       fprintf(stdout,"\n\nBye!\n\n");
       in = getch();
       return( rc );
}
```

Related Functions

The following functions are related to connections, drivers, and data sources. For more information about these functions, see Chapter 12, "INFORMIX-CLI Function Reference."

Function	Description
SQLDataSources	Retrieves a list of available data sources. INFORMIX-CLI retrieves this information from the initialization files. An application can present this information to a user or automatically select a data source.
SQLDrivers	Retrieves a list of installed drivers and their attributes. INFORMIX-CLI retrieves this information from the **odbcinst.ini** file. An application can present this information to a user or automatically select a driver.
SQLGetFunctions	Retrieves functions supported by a driver. This function allows an application to determine at runtime whether a particular function is supported by a driver.
SQLGetInfo	Retrieves general information about a driver and data source, including filenames, versions, conformance levels, and capabilities.
SQLGetTypeInfo	Retrieves the SQL data types supported by a driver and data source.
SQLSetConnectOption **SQLGetConnectOption**	Sets or retrieves connection options, such as the data-source access mode, automatic transaction commitment, time-out values, function tracing, data-translation options, and transaction isolation.

Chapter

Executing SQL Statements

5

This chapter describes the different ways an application can execute SQL statements.

SQL Statements and INFORMIX-CLI

An application can submit any SQL statement that is supported by a data source, and ODBC defines a standard syntax for SQL statements. For maximum interoperability, an application should submit only SQL statements that use this syntax; the driver translates these statements to the syntax that the data source uses. If an application submits an SQL statement that does not use the ODBC syntax, the driver passes the statement directly to the data source.

Important: *For CREATE TABLE and ALTER TABLE statements, applications should use the data type name returned by* ***SQLGetTypeInfo*** *in the TYPE_NAME column rather than the data type name defined in the SQL syntax.*

Use **SQLExecDirect** to execute a statement once. Use **SQLPrepare** to prepare a statement and execute it multiple times with **SQLExecute**. An application calls **SQLTransact** to commit or roll back a transaction.

Figure 5-1 illustrates a simple sequence of function calls to execute SQL statements.

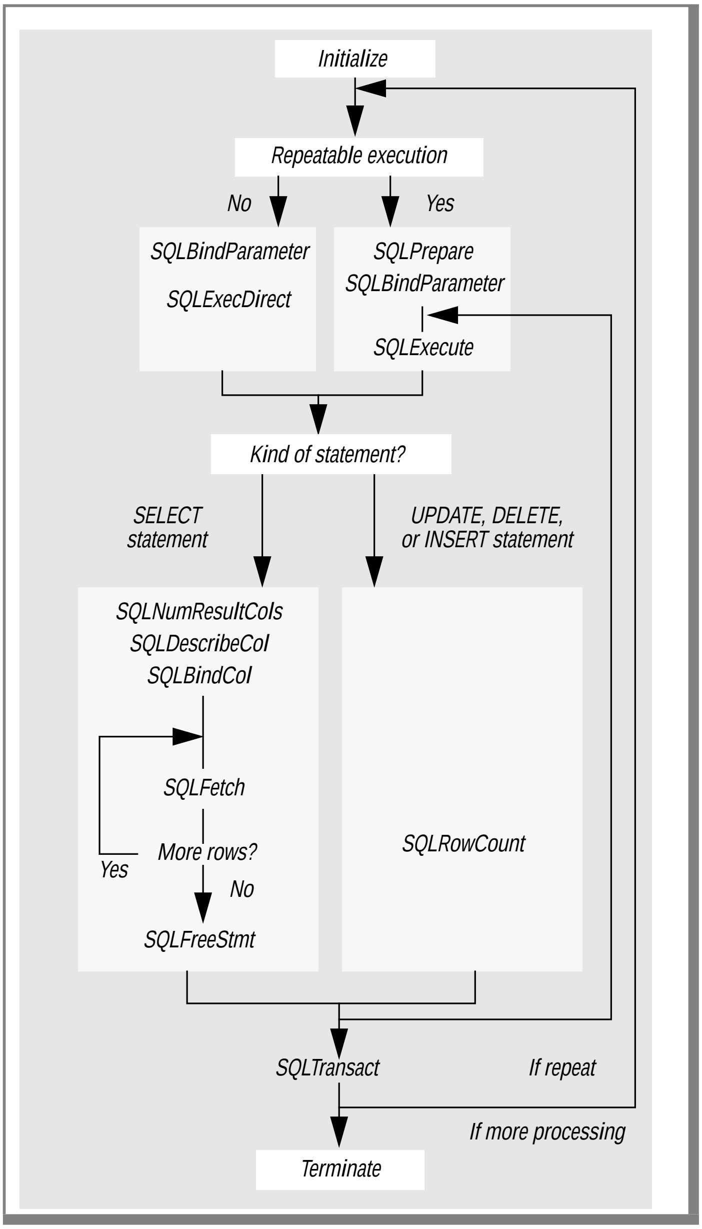


Figure 5-1
A Sequence of Function Calls to Execute SQL Statements

Allocating a Statement Handle

Before your application can submit an SQL statement, it must allocate a statement handle for the statement. To allocate a statement handle, an application performs the following operations:

1. Declares a variable of type HSTMT

   For example, the application could use the following declaration:

   ```
   HSTMT hstmt1;
   ```

2. Calls **SQLAllocStmt** and passes it the address of the variable and the connection handle (*hdbc*) with which to associate the statement

   The driver allocates memory to store information about the statement, associates the statement handle with the connection handle (*hdbc*), and returns the statement handle in the variable.

Executing an SQL Statement

Your application can submit an SQL statement for execution in the following ways:

- *Prepared* statements call **SQLPrepare** and then call **SQLExecute**.
- *Direct* statements call **SQLExecDirect**.

These options are similar, but not identical, to the prepared and immediate options in embedded SQL. For a comparison of the ODBC functions and embedded SQL, see Appendix C.

Prepared Execution

Prepare a statement before you execute it if either of the following conditions is true:

- The application can execute the statement more than once, possibly with intermediate changes to parameter values.
- The application needs information about the result set prior to execution.

A prepared statement executes faster than an unprepared statement because the data source compiles the statement, produces an access plan, and returns an access plan identifier to the driver. The data source minimizes processing time because it does not have to produce an access plan each time that it executes the statement. A prepared statement reduces network traffic because the driver sends the access plan identifier, instead of the entire statement, to the data source.

Important: *Committing or rolling back a transaction, either by calling* ***SQLTransact*** *or by using the SQL_AUTOCOMMIT connection option, can cause the data source to delete the access plans for all of the statement handles that belong to a connection handle. For more information, see "SQL_CURSOR_COMMIT_BEHAVIOR" and "SQL_CURSOR_ROLLBACK_BEHAVIOR" on page 12-196.*

To prepare and execute an SQL statement, an application performs the following operations:

1. Calls **SQLPrepare** to prepare the statement
2. Sets the values of any statement parameters

   For more information, see "Setting Parameter Values" on page 5-7.
3. Retrieves information about the result set, if necessary

   For more information, see "Determining the Characteristics of a Result Set" on page 6-5.
4. Calls **SQLExecute** to execute the statement
5. Repeats steps 2 through 4 as necessary

Direct Execution

Execute a statement directly if both of the following conditions are true:

- The application executes the statement only once.
- The application does not need information about the result set prior to execution.

To execute an SQL statement directly, an application performs the following operations:

1. Sets the values of any statement parameters

   For more information, see “Setting Parameter Values.”
2. Calls **SQLExecDirect** to execute the statement

Setting Parameter Values

An SQL statement can use parameter markers to contain values that the driver retrieves from the application at execution time. For example, an application might use the following statement to insert a row of data into the **EMPLOYEE** table:

```
INSERT INTO EMPLOYEE (NAME, AGE, HIREDATE) VALUES (?, ?, ?)
```

An application uses parameter markers instead of literal values when one of the following conditions occurs:

- It needs to execute the same prepared statement several times with different parameter values.
- The parameter values are unknown when the statement is prepared.
- The parameter values need to be converted from one data type to another.

To set a parameter value, an application performs the following operations:

1. Calls **SQLBindParameter** to bind a storage location to a parameter marker

   It also uses **SQLBindParameter** to specify:

   - the data types of the storage location and the column associated with the parameter.
   - precision and scale for the parameter.
2. Places the value of the parameter in the storage location

The application can perform these steps before or after a statement is prepared, but it must perform them before a statement executes.

Parameter values must be placed in storage locations in the C data types specified in **SQLBindParameter**, as the following table shows.

Parameter Value	SQL Data Type	C Data Type	Stored Value
ABC	SQL_CHAR	SQL_C_CHAR	ABC\0 [a]
10	SQL_INTEGER	SQL_C_SLONG	10
10	SQL_INTEGER	SQL_C_CHAR	10\0 [a]
1 P.M.	SQL_TIME	SQL_C_TIME	13,0,0 [b]
1 P.M.	SQL_TIME	SQL_C_CHAR	{t '13:00:00'}\0[a,c]

[a] The \0 value represents a null-termination byte; the null-termination byte is required only if the parameter length is SQL_NTS.

[b] The numbers in this list are the numbers stored in the fields of the TIME_STRUCT structure.

[c] The string uses the ODBC date escape clause.

Storage locations remain bound to parameter markers until the application calls **SQLFreeStmt** with the SQL_RESET_PARAMS option or the SQL_DROP option. An application can bind a different storage area to a parameter marker at any time by calling **SQLBindParameter**. An application can also change the value in a storage location at any time. When a statement is executed, the driver uses the current values in the most recently defined storage locations.

Performing Transactions

In *auto-commit* mode, every SQL statement is a complete transaction that is committed automatically. In *manual-commit* mode, a transaction consists of one or more statements. In manual-commit mode, when an application submits an SQL statement and no transaction is open, the driver implicitly begins a transaction. The transaction remains open until the application commits or rolls the transaction back with **SQLTransact**.

For INFORMIX-CLI, the default transaction mode is auto-commit. An application calls **SQLSetConnectOption** to switch between manual-commit and auto-commit mode. If an application switches from manual-commit to auto-commit mode, the driver commits any open transactions on the connection.

Applications should call **SQLTransact** to commit or roll back a transaction, rather than submitting a COMMIT or ROLLBACK statement. The result of a COMMIT or ROLLBACK statement depends on the driver and its associated data source.

Important: *Committing or rolling back a transaction, either by calling* ***SQLTransact*** *or by using the SQL_AUTOCOMMIT connection option, can cause the data source to close the cursors and delete the access plans for all of the statement handles that belong to a connection handle.*

For more information, see "SQL_CURSOR_COMMIT_BEHAVIOR" and "SQL_CURSOR_ROLLBACK_BEHAVIOR" on page 12-196.

Retrieving Information About the Data-Source Catalog

The functions listed in "Retrieving Information About Data-Source System Tables" on page 11-8 are catalog functions that return information about a data-source catalog. Each function returns the information as a result set. Use **SQLBindCol** and **SQLFetch** to retrieve these results.

Sending Parameter Data at Execution Time

Use the following functions to send parameter data, for instance, for parameters of the SQL_LONGVARCHAR or SQL_LONGVARBINARY types, when a statement executes:

- **SQLBindParameter**
- **SQLParamData**
- **SQLPutData**

To indicate that your application plans to send parameter data when the statement executes, call **SQLBindParameter**, and set the *pcbValue* buffer for the parameter to the result of the SQL_LEN_DATA_AT_EXEC(*length*) macro. If the *fSqlType* argument is SQL_LONGVARBINARY or SQL_LONGVARCHAR and the driver returns Y for the SQL_NEED_LONG_DATA_LEN information type in **SQLGetInfo**, then *length* is the total number of bytes of data to be sent for the parameter; otherwise, it is ignored.

Set the *rgbValue* argument to a value that can be used to retrieve the data at runtime. For example, *rgbValue* might point to a storage location that contains the data when the statement executes or to a file that contains the data. The driver returns the value to the application when the statement executes.

When the driver processes a call to **SQLExecute** or **SQLExecDirect** and the executing statement includes a data-at-execution parameter, the driver returns SQL_NEED_DATA.

To send the parameter data, an application performs the following operations:

1. Calls **SQLParamData**, which returns *rgbValue* (as set with **SQLBindParameter**) for the first data-at-execution parameter
2. Calls **SQLPutData** one or more times to send data for the parameter

   More than one call is needed if the data value is larger than the buffer; multiple calls are allowed only if the C data type is character or binary and the SQL data type is character, binary, or data-source specific.
3. Calls **SQLParamData** again to indicate that all the data has been sent for the parameter

   If another data-at-execution parameter remains, the driver returns *rgbValue* for that parameter and SQL_NEED_DATA for the function return code. Otherwise, it returns SQL_SUCCESS for the function return code.
4. Repeats steps 2 and 3 for the remaining data-at-execution parameters

For additional information, see "SQLBindParameter" on page 12-25.

Specifying Arrays of Parameter Values

To specify multiple sets of parameter values for a single SQL statement, an application calls **SQLParamOptions**. For example, if there are 10 sets of column values to insert into a table, and the same SQL statement can be used for all 10 operations, the application can set up an array of values and then submit a single INSERT statement.

If an application uses **SQLParamOptions**, it must allocate enough memory to handle the array of values.

To retrieve a list of the procedures stored in a data source, an application calls **SQLProcedures**.

Related Functions

INFORMIX-CLI provides the following functions related to SQL statements. For more information about these functions, see Chapter 12.

Function	Description
SQLNativeSql	Retrieves the SQL statement as processed by the data source, with escape sequences translated to SQL code used by the data source.
SQLNumParams	Retrieves the number of parameters in an SQL statement.
SQLSetStmtOption **SQLSetConnectOption** **SQLGetStmtOption**	Sets or retrieves statement options, such as orientation for binding row sets, maximum amount of variable-length data to return, maximum number of result set rows to return, and query time-out value. **SQLSetConnectOption** sets options for all the statements in a connection.

Chapter

Retrieving Results

6

This chapter describes the different ways that an application can retrieve results.

Result Sets and INFORMIX-CLI

A SELECT statement is used to retrieve data that meets a given set of specifications. In the following example, a SELECT statement retrieves all columns for employees named Jones from all the rows in the **EMPLOYEE** table:

```
SELECT * FROM EMPLOYEE WHERE EMPNAME = "Jones"
```

INFORMIX-CLI functions can also retrieve data. For example, **SQLColumns** retrieves data about columns in the data source. These sets of data, called result sets, can contain zero or more rows.

Other SQL statements, such as GRANT or REVOKE, do not return result sets. For these statements, the return code from **SQLExecute** or **SQLExecDirect** is usually the only source of information about whether the statement is successful. For INSERT, UPDATE, and DELETE statements, an application can call **SQLRowCount** to return the number of affected rows.

The steps that an application takes to process a result set depend on what is known about it. The following table defines known and unknown result sets.

Type of Result Set	Definition	Example
Known	The application knows the exact form of the SQL statement and, therefore, the result set when the statement compiles.	For example, the following query returns two specific columns: `SELECT EMPNO, EMPNAME FROM EMPLOYEE`
Unknown	The application does not know the exact form of the SQL statement or, therefore, the result set when the statement compiles.	For example, the following ad hoc query returns all the currently defined columns in the EMPLOYEE table: `SELECT * FROM EMPLOYEE` The application might not predict the format of these results before it executes the command.

Assigning Storage for Results (Binding)

An application can assign storage for results before or after it executes an SQL statement. If an application prepares or executes the SQL statement first, it can inquire about the result set before it assigns storage for results. For example, if the result set is unknown, the application must retrieve the number of columns in the result set before it can assign storage for them.

To associate storage for a column of data, an application calls **SQLBindCol** and passes it the following information:

- The data type to which the data is to be converted

  For more information, see Appendix B.
- The address of an output buffer for the data

  The application must allocate this buffer, which must be large enough to hold the data in its converted form.
- The length of the output buffer

  This value is ignored if the returned data has a fixed width in C, such as an integer, real number, or date structure.
- The address of a storage buffer in which to return the number of bytes of available data

Determining the Characteristics of a Result Set

To determine the characteristics of a result set, an application can perform the following actions:

- Call **SQLNumResultCols** to determine how many columns a request returned
- Call **SQLColAttributes** or **SQLDescribeCol** to describe a column in the result set

If the result set is unknown, an application can use the information from these functions to bind the columns in the result set. An application can call these functions at any time after a statement is prepared or executed.

Tip: *For optimal performance, an application should call* ***SQLColAttributes****,* ***SQLDescribeCol****, and* ***SQLNumResultCols*** *after a statement executes. In data sources that emulate statement preparation, these functions sometimes execute more slowly before a statement executes because the information that these functions return is not readily available until after the statement has executed.*

Fetching Result Data

You can perform several operations to retrieve data once the characteristics of the result set are known. To retrieve a row of data from the result set, an application performs the following operations:

1. Calls **SQLBindCol** to bind the columns of the result set to storage locations if it has not already done so
2. Calls **SQLFetch** to move to the next row in the result set and retrieve data for all bound columns

Figure 6-1 shows the operations that an application uses to retrieve data from the result set.

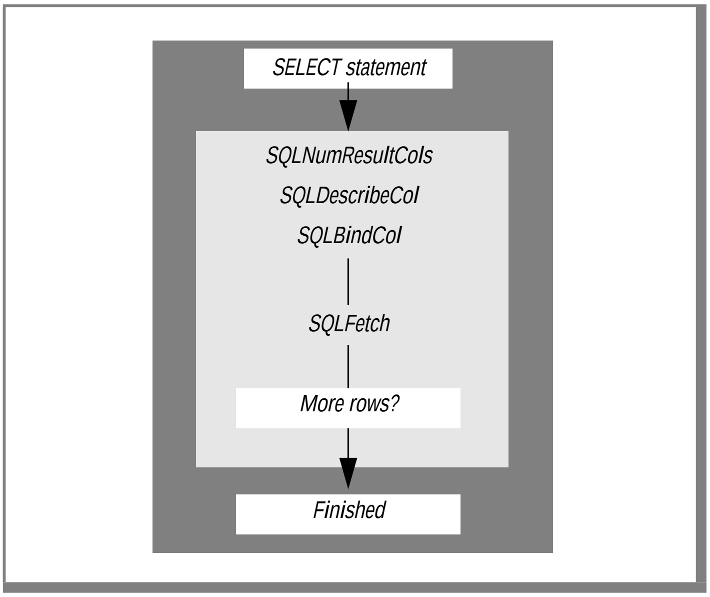


Figure 6-1
Operations That an Application Uses to Retrieve Data from the Result Set

Using Cursors

The INFORMIX-CLI driver maintains a cursor to indicate the current position in the result set, much as the cursor on a computer screen indicates the current position.

You can use **SQLFetch** or **SQLExtendedFetch** to retrieve data from a result set. Each time that an application calls **SQLFetch**, the driver moves the cursor to the next row and returns that row. To retrieve a row of data that has already been retrieved from the result set, the application must close the cursor by calling **SQLFreeStmt** with the SQL_CLOSE option, re-execute the SELECT statement, and fetch rows with **SQLFetch** until the target row is retrieved.

Important: *Committing or rolling back a transaction, either by calling* ***SQLTransact*** *or by using the SQL_AUTOCOMMIT connection option, can cause the data source to close the cursors for all hstmts on an hdbc.*

For more information, see "SQL_CURSOR_COMMIT_BEHAVIOR" and "SQL_CURSOR_ROLLBACK_BEHAVIOR" on page 12-196.

Retrieving Data from Unbound Columns

To retrieve data from unbound columns (that is, columns for which storage has not been assigned with **SQLBindCol**), an application uses **SQLGetData**. The application first calls **SQLFetch** or **SQLExtendedFetch** to position the cursor on the next row. It then calls **SQLGetData** to retrieve data from specific unbound columns.

An application can retrieve data from bound and unbound columns in the same row. It calls **SQLBindCol** to bind as many columns as you specify. It calls **SQLFetch** or **SQLExtendedFetch** to position the cursor on the next row of the result set and to retrieve all bound columns. It then calls **SQLGetData** to retrieve data from unbound columns.

If the column data type is character, binary, or data-source specific and the column contains more data than can be retrieved in a single call, an application might call **SQLGetData** more than once for that column, as long as the data is being transferred to a buffer of type SQL_C_CHAR or SQL_C_BINARY. For example, data of the SQL_LONGVARBINARY and SQL_LONGVARCHAR types might need to be retrieved in several parts.

The INFORMIX-CLI driver can return data with **SQLGetData** for both bound and unbound columns in any order. For maximum interoperability, however, your application should call **SQLGetData** only for columns to the right of the right-most bound column and then only in left-to-right order.

Assigning Storage for Row Sets (Binding)

In addition to binding individual rows of data, an application can call **SQLBindCol** to assign storage for a *row set* (one or more rows of data). To specify how many rows of data are in a row set, an application calls **SQLSetStmtOption** with the SQL_ROWSET_SIZE option.

By default, row sets are bound in column-wise fashion. They can also be bound in row-wise fashion.

To assign storage for column-wise binding

1. Allocate an array of data storage buffers.

   The array has as many elements as there are rows in the row set.
2. Allocate an array of storage buffers to hold the number of bytes that are available to return for each data value.

   The array has as many elements as there are rows in the row set.
3. Call **SQLBindCol** and specify the address of the data array, the size of one element of the data array, the address of the number-of-bytes array, and the type to which the data will be converted.

   When data is retrieved, the driver uses the array element size to determine where to store successive rows of data in the array.

To assign storage for row-wise binding

1. Declare a structure that can hold a single row of retrieved data and the associated data lengths.

   To bind each column, the structure includes one field to contain data and one field to contain the number of bytes of data that are available to return.
2. Allocate an array of these structures.

   This array has as many elements as there are rows in the row set.
3. Call **SQLBindCol** for each column to be bound.
4. In each call, specify the address of the column data field in the first array element, the size of the data field, the address of the column number-of-bytes field in the first array element, and the type to which the data will be converted.
5. Call **SQLSetStmtOption** with the SQL_BIND_TYPE option and specify the size of the structure.

   When the data is retrieved, the driver uses the structure size to determine where to store successive rows of data in the array.

Retrieving Row Set Data

Before it retrieves row set data, an application calls **SQLSetStmtOption** with the SQL_ROWSET_SIZE option to specify the number of rows in the row set. It then binds columns in the row set with **SQLBindCol**. The row set is bound using a column- or row-wise method. For more information, see "Assigning Storage for Row Sets (Binding)" on page 6-7.

To retrieve row set data, an application calls **SQLExtendedFetch**.

For maximum interoperability, an application should not use **SQLGetData** to retrieve data from unbound columns in a block of data (more than one row) that has been retrieved with **SQLExtendedFetch**. To determine whether a driver can return data with **SQLGetData** from a block of data, an application calls **SQLGetInfo** with the SQL_GETDATA_EXTENSIONS option.

Using Block and Scrollable Cursors

Cursors in SQL scroll forward through a result set, returning one row at a time. However, interactive applications often require forward and backward scrolling, absolute or relative positioning within the result set, and the ability to retrieve and update blocks of data, or *row sets*.

A *block* cursor attribute allows an application to retrieve and update row set data. A *scrollable* cursor attribute allows an application to scroll forward or backward through the result set or move to an absolute or relative position in the result set. Cursors can have one or both attributes.

Block Cursors

An application calls **SQLSetStmtOption** with the SQL_ROWSET_SIZE option to specify the row set size. The application can call **SQLSetStmtOption** to change the row set size at any time. Each time that the application calls **SQLExtendedFetch**, the driver returns the next *row-set-size* rows of data. After the data is returned, the cursor points to the first row in the row set. By default, the row set size is one.

Scrollable Cursors

Applications have different needs to sense changes in the tables underlying a result set. For example, when balancing financial data, an accountant needs data that appears static because it is impossible to balance books when the data changes continually. When selling concert tickets, a clerk needs up-to-the minute, or dynamic, data on which tickets are still available. Various cursor models are designed to meet these needs. Each requires different sensitivities to changes in the tables that underlie the result set.

Static Cursors

In *static* cursors, the data in the underlying tables appears to be static. The membership, order, and values in the result set used by a static cursor are generally fixed when the cursor is opened. Rows updated, deleted, or inserted by other users, including other cursors in the same application, are not detected by the cursor until it closes and then reopens. The SQL_STATIC_SENSITIVITY information type returns whether the cursor can detect rows that it has updated, deleted, or inserted.

Static cursors are commonly implemented by taking a snapshot of the data or locking the result set. In the former case, the cursor diverges from the underlying tables as other users make changes; in the latter case, other applications and cursors in the same application cannot change the data.

Dynamic Cursors

In *dynamic* cursors, the data appears to be dynamic. The membership, order, and values in the result set that a dynamic cursor uses are always changing. When data is fetched, the cursor detects rows updated, deleted, or inserted by all users (the cursor, other cursors in the same application, and other applications). Although dynamic cursors are ideal for many situations, they are difficult to implement.

Key-Set-Driven Cursors

Key-set-driven cursors possess some attributes of static and dynamic cursors. Like static cursors, the membership and ordering of the result set of a key-set-driven cursor is generally fixed when the cursor opens. As with dynamic cursors, most changes to the values in the underlying result set are visible to the cursor when data is fetched.

Specifying the Cursor Type

To specify the cursor type, an application calls **SQLSetStmtOption** with the SQL_CURSOR_TYPE option. The application can specify a cursor that scrolls forward, a static cursor, or a dynamic cursor. If the application specifies a mixed cursor, it also specifies the size of the key set that the cursor uses.

To use the ODBC cursor library, an application calls **SQLSetConnectOption** with the SQL_ODBC_CURSORS option before it connects to the data source. For more information on the ODBC cursor library, see the *Microsoft ODBC Programmer's Reference and SDK Guide,* Version 2.0.

An application calls **SQLExtendedFetch** to scroll the cursor backward, forward, or to an absolute or relative position in the result set.

Specifying Cursor Concurrency

Concurrency is the ability of more than one user to access the same data at the same time. A transaction is *serializable* if it is performed so that it appears as though no other transactions operate on the same data at the same time. Suppose one transaction doubles data values and another adds 1 to data values. If the transactions are serializable, and both attempt to operate on the values 0 and 10 at the same time, the final values are 1 and 21 or 2 and 22, depending on which transaction occurs first. If the transactions are not serializable, the final values are 1 and 21, 2 and 22, 1 and 22, or 2 and 21; the sets of values 1 and 22, and 2 and 21 are the result of the transactions acting on each value in a different order.

To maintain database integrity, you must to be able to serialize a transaction. However, a serialized transaction locks the result set which prohibits cursor concurrency.

Positioned Update and Delete Statements

To modify data in the result set, an application can update or delete the row to which the cursor currently points. Such an operation is known as a positioned update or delete statement.

INFORMIX-CLI positioned update and delete statements are similar to such statements in embedded SQL. After an application executes a SELECT statement to create a result set, it calls **SQLFetch** one or more times to position the cursor on the row to be updated or deleted. To update or delete the row, the application then executes an SQL statement with the following syntax on a different *hstmt*:

```
UPDATE table-name
SET column-identifier = {expression | NULL}
[, column-identifier = {expression | NULL}]...
WHERE CURRENT OF cursor-name
DELETE FROM table-name WHERE CURRENT OF cursor-name
```

Positioned update and delete statements require cursor names. An application can name a cursor with **SQLSetCursorName**. If the application has not named the cursor by the time that the driver executes a SELECT statement, the driver generates a cursor name. To retrieve the cursor name for an *hstmt*, an application calls **SQLGetCursorName**.

To execute a positioned update or delete statement, an application must follow these guidelines:

- The SELECT statement that creates the result set must use a FOR UPDATE clause.
- The cursor name used in the UPDATE or DELETE statement must be the same as the cursor name associated with the SELECT statement.
- The application must use different *hstmts* for the SELECT statement and the UPDATE or DELETE statement.
- The *hstmts* for the SELECT statement and the UPDATE or DELETE statement must be on the same connection.

To determine whether a data source supports positioned update and delete statements, an application calls **SQLGetInfo** with the SQL_POSITIONED_STATEMENTS option.

Processing Multiple Results

SELECT statements return result sets. UPDATE, INSERT, and DELETE statements return a count of affected rows. If any of these statements are in procedures, are batched, or are submitted with arrays of parameters, they can return multiple result sets or counts.

To process a batch of statements, a statement with arrays of parameters, or a procedure that returns multiple result sets or row counts, an application performs the following operations:

1. Calls **SQLExecute** or **SQLExecDirect** to execute the statement or procedure
2. Calls **SQLRowCount** to determine the number of rows affected by an UPDATE, INSERT, or DELETE statement

   For statements or procedures that return result sets, the application calls functions to determine the characteristics of the result set and retrieve data from the result set.
3. Calls **SQLMoreResults** to determine whether another result set or row count is available
4. Repeats steps 2 and 3 until **SQLMoreResults** returns SQL_NO_DATA_FOUND

Chapter

7 Retrieving Status and Error Information

This chapter defines INFORMIX-CLI return codes and error-handling protocol. The return codes indicate whether a function succeeded, succeeded but returned a warning, or failed. The error-handling protocol defines how the components in an INFORMIX-CLI connection construct and return error messages through **SQLError**.

The protocol describes the following points:

- Use of the error text to identify the source of an error
- Rules to ensure consistent and useful error information
- Responsibility for setting the SQLSTATE based on the native error

Function Return Codes

When an INFORMIX-CLI application calls a function, the driver executes the function and returns a predefined code. These return codes indicate success, warning, or failure status. The following table defines the return codes.

Return Code	Description
SQL_SUCCESS	Function completed successfully; no additional information is available.
SQL_SUCCESS_WITH_INFO	Function completed successfully, possibly with a nonfatal error. The application can call **SQLError** to retrieve additional information.
SQL_NO_DATA_FOUND	All rows from the result set have been fetched, but no data was found.
SQL_ERROR	Function failed. The application can call **SQLError** to retrieve error information.

(1 of 2)

Return Code	Description
SQL_INVALID_HANDLE	Function failed due to an invalid environment handle, connection handle, or statement handle. This situation indicates a programming error. No additional information is available from **SQLError**.
SQL_STILL _EXECUTING	Function is still busy processing an asynchronous request.
SQL_NEED_DATA	While the driver was processing a statement, it determined that the application needs to send parameter data values.

(2 of 2)

The application is responsible for taking the appropriate action based on the return code.

Error Messages

INFORMIX-CLI defines a layered architecture to connect an application to a data source. **SQLError** returns error messages that follow the standard INFORMIX-CLI format. An error message not only explains the error but also provides the identity of the component in which it occurred. Because **SQLError** does not return the identity of the component in which the error occurred, this information is embedded in the error text.

Error Text Format

Error messages that **SQLError** returns come from two sources: data sources and INFORMIX-CLI. Consequently, the error text that **SQLError** returns has two formats: one for errors that occur in a data source and one for errors that occur in INFORMIX-CLI.

If INFORMIX-CLI receives an error message from a data source, it identifies the data source as the source of the error. It also identifies itself as the component that received the error. For errors that occur in a data source, the error text uses the following format:

```
[vendor_identifier][INFORMIX-CLI_component_identifier]
[data_source_identifier]data_source_supplied_text
```

If the source of an error is INFORMIX-CLI, the error message explains which INFORMIX-CLI component caused the error. For errors that do not occur in a data source, the error text uses the following format:

```
[vendor_dentifier][INFORMIX-CLI _component_identifier]
component_supplied_text
```

The following table shows the meaning of each element.

Element	Meaning
vendor_identifier	Identifies the vendor of the component in which the error occurred or the vendor of the component that received the error directly from the data source.
INFORMIX-CLI_component _identifier	Identifies the component in which the error occurred or that received the error directly from the data source.
data_source_identifier	Identifies the data source. For multiple-tier drivers, this value is the DBMS product.
component_supplied_text	Error text that INFORMIX-CLI generates.
data_source_supplied_text	Error text that the data source generates.

The brackets ([ ]) are included in the error text. They do not indicate optional items.

Sample Error Messages

The following examples show how various components in a connection might generate the text of error messages and how INFORMIX-CLI might return the error messages to the application with **SQLError**.

Sample Error Returned from the Driver

INFORMIX-CLI sends requests to a data source and returns information to the application. If the INFORMIX-CLI architecture includes a driver manager, the INFORMIX-CLI driver formats and returns arguments for **SQLError** because it is the component that interfaces with the driver manager.

For example, if an INFORMIX-CLI driver for INFORMIX-SE encounters a duplicate cursor name, it might return the following arguments for **SQLError**:

```
szSQLState = "3C000"
pfNativeError = NULL
szErrorMsg = "[Informix][ODBC Informix Driver]
Duplicate cursor name:EMPLOYEE_CURSOR."
pcbErrorMsg = 67
```

Because the error occurred in the driver, it adds prefixes to the error text for the vendor (Informix) and the driver (ODBC Informix Driver).

Sample Error Returned from the Driver Manager

The driver manager can also generate error messages. For example, if an application passed an invalid argument value to **SQLDataSources**, the driver manager might format and return the following arguments for **SQLError**:

```
szSQLState = "S1009"
pfNativeError = NULL
szErrorMsg = "[Informix][ODBC DLL]Invalid argument value:SQLDataSources."
pcbErrorMsg = 60
```

Because the error occurred in the driver manager, it adds prefixes to the error text for its vendor (Informix) and its identifier (ODBC DLL).

Sample Error Returned from the Data Source

If the data source could not find the table **EMPLOYEE**, the driver might format and return the following arguments for **SQLError**:

```
szSQLState = "S0002"
pfNativeError = -206
szErrorMsg = "[Informix][ODBC Informix Driver][Informix]The
specified table (EMPLOYEE) is not in the database."
pcbErrorMsg = 96
```

Because the error occurred in the data source, the driver added a prefix for the data source identifier (Informix) to the error text. Because the driver component interfaced with the data source, it adds prefixes for its vendor (Informix) and identifier (ODBC Informix Driver) to the error text. For a description of an error that the data source returns, look up the error message number in the *Informix Error Messages* manual.

Processing Error Messages

You can provide your users with the error information that **SQLError** returns: the SQLSTATE, the error message number, the error message, and the corrective action. You must parse the text to separate the error message from the corrective action. You need to take the appropriate action based on the error or provide the user with a choice of actions.

Retrieving Error Messages

If a function other than **SQLError** returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an application can call **SQLError** to obtain additional information. The application might need to call **SQLError** more than once to retrieve all the error messages from a function because a function might return more than one error message. When the application calls a different function, the error messages from the previous function are deleted.

Additional error or status information can come from one of the following sources:

- Error or status information from an ODBC function, indicating that a programming error was detected
- Error or status information from the data source, indicating that an error occurred during SQL statement processing

The information that **SQLError** returns is in the same format as that provided by SQLSTATE in the X/Open and SQL Access Group SQL CAE specification (1992). **SQLError** never returns error information about itself.

Chapter

8 Terminating Transactions and Connections

The INFORMIX-CLI interface provides functions to terminate statements, transactions, and connections, as well as to free statement (*hstmt*), connection (*hdbc*), and environment (*henv*) handles.

Terminating Statement Processing

To free the resources associated with a statement handle, an application calls **SQLFreeStmt**. The **SQLFreeStmt** function has the following options:

- SQL_CLOSE

  This option closes the cursor, if one exists, and discards pending results. The application can use the statement handle again later.
- SQL_DROP

  This option closes the cursor, if one exists, discards pending results, and frees all resources associated with the statement handle.
- SQL_UNBIND

  This option frees all return buffers bound by **SQLBindCol** for the statement handle.
- SQL_RESET_PARAMS

  This option frees all parameter buffers requested by **SQLBindParameter** for the statement handle.

Terminating Transactions

An application calls **SQLTransact** to commit or roll back the current transaction.

Terminating Connections

To terminate a connection to a driver or to a data source, an application closes the connection associated with a connection handle and frees the connection and environment handles. To terminate a connection to a driver and data source, an application performs the following operations:

1. It calls **SQLDisconnect** to close the connection.

   You can then use the handle to reconnect to the same data source or to a different data source.
2. It calls **SQLFreeConnect** to free the connection handle and free all resources associated with the handle.
3. It calls **SQLFreeEnv** to free the environment handle and free all resources associated with the handle.

Chapter

Constructing an INFORMIX-CLI Application

9

This chapter provides the following examples of C-language source code for INFORMIX-CLI-enabled applications:

- An example that uses static SQL functions to create a table, add data to it, and select the inserted data
- An example of interactive, ad-hoc query processing

Static SQL Example

The following example constructs SQL statements within the application. The example comments include equivalent embedded SQL calls for illustrative purposes.

```
#include "sql.h"
#include <string.h>

#ifndef NULL
#define NULL 0
#endif
#define MAX_NAME_LEN 50
#define MAX_STMT_LEN 100
int print_err(HDBC hdbc, HSTMT hstmt);

int example1(server, uid, pwd)
UCHAR * server;
UCHAR * uid;
UCHAR * pwd;
{
HENV   henv;
HDBC   hdbc;
HSTMT  hstmt;

SDWORD  id;
UCHAR   name[MAX_NAME_LEN + 1];
UCHAR   create[MAX_STMT_LEN]
UCHAR   insert[MAX_STMT_LEN]
UCHAR   select[MAX_STMT_LEN]
SDWORD  namelen;
RETCODE rc;

/* EXEC SQL CONNECT TO :server USER :uid USING :pwd; */
/* Allocate an environment handle.                   */
/* Allocate a connection handle.                     */
/* Connect to a data source.                         */
/* Allocate a statement handle.                      */

SQLAllocEnv(&henv);
SQLAllocConnect(henv, &hdbc);
rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(print_err(hdbc, SQL_NULL_HSTMT));
SQLAllocStmt(hdbc, &hstmt);

/* EXEC SQL CREATE TABLE NAMEID (ID integer, NAME varchar(50)); */
/* Execute the SQL statement.                                   */

lstrcpy(create, "CREATE TABLE NAMEID (ID INTEGER, NAME
    VARCHAR(50))");
rc = SQLExecDirect(hstmt, create, SQL_NTS);
```

```
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(print_err(hdbc, hstmt));

/* EXEC SQL COMMIT WORK;      */
/* Commit the table creation. */

/* Note that the default transaction mode for drivers that support */
/* SQLSetConnectOption is auto-commit and SQLTransact has no effect */

SQLTransact(hdbc, SQL_COMMIT);

/* EXEC SQL INSERT INTO NAMEID VALUES ( :id, :name ); */
/* Show the use of the SQLPrepare/SQLExecute method:  */
/* Prepare the insertion and bind parameters.         */
/* Assign parameter values.                           */
/* Execute the insertion.                             */

lstrcpy(insert, "INSERT INTO NAMEID VALUES (?, ?)");
if (SQLPrepare(hstmt, insert, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER,

0, 0, &id, 0, NULL);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,

MAX_NAME_LEN, 0, name, 0, NULL);
id=500;
lstrcpy(name, "Babbage");
if (SQLExecute(hstmt) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));

/* EXEC SQL COMMIT WORK; */
/* Commit the insertion. */

SQLTransact(hdbc, SQL_COMMIT);

/* EXEC SQL DECLARE c1 CURSOR FOR SELECT ID, NAME FROM NAMEID; */
/* EXEC SQL OPEN c1;                                           */
/* Show the use of the SQLExecDirect method.                   */
/* Execute the selection.                                      */
/* Note that the application does not declare a cursor.  */

lstrcpy(select, "SELECT ID, NAME FROM NAMEID");
if (SQLExecDirect(hstmt, select, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));

/* EXEC SQL FETCH c1 INTO :id, :name;                 */
/* Bind the columns of the result set with SQLBindCol. */
/* Fetch the first row.                               */

SQLBindCol(hstmt, 1, SQL_C_SLONG, &id, 0, NULL);
SQLBindCol(hstmt, 2, SQL_C_CHAR, name, (SDWORD)sizeof(name), &namelen);
SQLFetch(hstmt);

/* EXEC SQL COMMIT WORK;   */
/* Commit the transaction. */
```

```
SQLTransact(hdbc, SQL_COMMIT);

/* EXEC SQL CLOSE c1;         */
/* Free the statement handle. */

SQLFreeStmt(hstmt, SQL_DROP);

/* EXEC SQL DISCONNECT;             */
/* Disconnect from the data source. */
/* Free the connection handle.      */
/* Free the environment handle.     */

SQLDisconnect(hdbc);
SQLFreeConnect(hdbc);
SQLFreeEnv(henv);

return(0);
}
```

Interactive Ad-Hoc Query Example

The following example illustrates how an application can determine the nature of the result set before it retrieves results:

```
#include "sql.h"
#include <string.h>
#include <stdlib.h>

#define MAXCOLS 100
#define max(a,b) (a>b?a:b)

int    print_err(HDBC hdbc, HSTMT hstmt);
UDWORD display_size(SWORD coltype, UDWORD collen, UCHAR *colname);

example2(server, uid, pwd, sqlstr)
UCHAR * server;
UCHAR * uid;
UCHAR * pwd;
UCHAR * sqlstr;
{
int     i;
HENV    henv;
HDBC    hdbc;
HSTMT   hstmt;
UCHAR   errmsg[256];
UCHAR   colname[32];
SWORD   coltype;
SWORD   colnamelen;
SWORD   nullable;
UDWORD  collen[MAXCOLS];
SWORD   scale;
SDWORD  outlen[MAXCOLS];
UCHAR * data[MAXCOLS];
SWORD   nresultcols;
SDWORD  rowcount;
RETCODE rc;

/* Allocate environment and connection handles. */
/* Connect to the data source.                  */
/* Allocate a statement handle.                 */
SQLAllocEnv(&henv);
SQLAllocConnect(henv, &hdbc);
rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(print_err(hdbc, SQL_NULL_HSTMT));
SQLAllocStmt(hdbc, &hstmt);
/* Execute the SQL statement. */
if (SQLExecDirect(hstmt, sqlstr, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));

/* See what kind of statement it was.If there are no result     */
/* columns,the statementis not a SELECT statement.If the      */
/* number of affected rows is greater than 0, the statement was   */
/* probably an UPDATE, INSERT, or DELETE statement, so print the  */
/* number of affected rows.  If the number of affected rows is 0, */
/* the statement is probably a DDL statement, so print that the   */
/* operation was successful and commit it.                        */

SQLNumResultCols(hstmt, &nresultcols);
if (nresultcols = = 0) {
    SQLRowCount(hstmt, &rowcount);
    if (rowcount > 0 ) {
        printf("%ld rows affected.\n", rowcount);
    } else {
        printf("Operation successful.\n");
    }
    SQLTransact(hdbc, SQL_COMMIT);

/* Otherwise, display the column names of the result set and use the */
/* display_size() function to compute the length needed by each data */
/* type.  Next, bind the columns and specify all data will be        */
/* converted to char.  Finally, fetch and print each row, printing   */
/* truncation messages as necessary.                                 */

} else {
    for (i = 0; i < nresultcols; i++) {
        SQLDescribeCol(hstmt, i + 1, colname, (SWORD)sizeof(colname),
                        &colnamelen, &coltype, &collen[i], &scale,
                        &nullable);
        collen[i] = display_size(coltype, collen[i], colname);
        printf("%*.*s", collen[i], collen[i], colname);
        data[i] = (UCHAR *) malloc(collen[i] + 1);
        SQLBindCol(hstmt, i + 1, SQL_C_CHAR, data[i], collen[i],
                    &outlen[i]);
    }
    while (TRUE) {
        rc = SQLFetch(hstmt);
        if (rc = = SQL_SUCCESS || rc = = SQL_SUCCESS_WITH_INFO) {
            errmsg[0] = '\0';
            for (i = 0; i < nresultcols; i++)
                if (outlen[i] = = SQL_NULL_DATA) {
                    lstrcpy(data[i], "NULL");
                } else if (outlen[i] >= collen[i]) {
                    sprintf(&errmsg[strlen(errmsg)],
                            "%d chars truncated, col %d\n",
                            *outlen[i] - collen[i] + 1,
                            colnum;
                }
                printf("%*.*s ", collen[i], collen[i], data[i]);
            }
            printf("\n%s", errmsg);
        } else {
            break;
        }
    }
}
```

```
/* Free the data buffers. */
for (i = 0; i < nresultcols; i++) {
    free(data[i]);
}

SQLFreeStmt(hstmt, SQL_DROP );  /* Free the statement handle.       */
SQLDisconnect(hdbc);            /* Disconnect from the data source. */
SQLFreeConnect(hdbc);           /* Free the connection handle.      */
SQLFreeEnv(henv);               /* Free the environment handle.     */

return(0);
}
/**********************************************************/
/* The following function is included for completeness, but */
/* is not relevant for understanding the function of ODBC.  */
/**********************************************************/

#define MAX_NUM_PRECISION 15

/* Define max length of char string representation of number as:      */
/*   =  max(precision) + leading sign + E + exp sign + max exp length */
/*   =  15             + 1            + 1 + 1        + 2              */
/*   =  15 + 5                                                        */

#define MAX_NUM_STRING_SIZE (MAX_NUM_PRECISION + 5)

UDWORD  display_size(coltype, collen, colname)
SWORD   coltype;
UDWORD  collen;
UCHAR * colname;
{
switch (coltype) {

    case SQL_CHAR:
    case SQL_VARCHAR:
        return(max(collen, strlen(colname)));

    case SQL_SMALLINT:
        return(max(6, strlen(colname)));

    case SQL_INTEGER:
        return(max(11, strlen(colname)));

    case SQL_DECIMAL:
    case SQL_NUMERIC:
    case SQL_REAL:
    case SQL_FLOAT:
    case SQL_DOUBLE:
        return(max(MAX_NUM_STRING_SIZE, strlen(colname)));

    /* Note that this function only supports the core data types. */
    default:
        printf("Unknown datatype, %d\n", coltype);
        return(0);
    }
```

Chapter

10

Designing Performance-Oriented Applications

This chapter provides guidelines for designing and coding performance-oriented INFORMIX-CLI applications. These guidelines are based on four general performance rules:

- Reduce network traffic as much as possible.
- Simplify queries as much as possible.
- Optimize the application-to-driver interaction.
- Limit disk I/O to improve performance.

Connecting to a Data Source

Connection management is extremely important to application performance because the process of connecting to a data source is expensive. To optimize your application, design it to connect only once. Rather than perform multiple connections, use multiple statement handles.

Making One Connection

Some ODBC applications are designed to call information-gathering routines that have no record of connection handles (*hdbc* pointers) that already exist. For example, an application might establish a connection and then call a routine in a separate DLL or shared library that reattaches and gathers up-front information about the driver to be used later in the application.

To optimize performance, applications that are designed as separate entities should pass the already-connected *hdbc* pointer to the data-collection routine instead of establishing a second connection.

Using Statement Handles

You can associate multiple statement handles with one connection handle. *Statement handles* provide memory storage for information about SQL statements. You should use statement handles to manage multiple SQL statements instead of connecting and disconnecting several times to execute the SQL statements.

Retrieving Information About a Data Source

Catalog functions are slow compared to all other INFORMIX-CLI functions. The functions listed in "Retrieving Information About Data-Source System Tables" on page 11-8 are catalog functions that return information about a data-source catalog. **SQLGetTypeInfo** is also a potentially expensive function and is included in this discussion of catalog functions.

Minimizing the Use of Catalog Functions

Although you probably need to use catalog functions to write an ODBC-compliant application, use them sparingly. To return the necessary result set for a single call to a catalog function, a driver might have to perform multiple queries, joins, subqueries, and unions. For this reason, frequent use of catalog functions in an application will likely result in poor performance.

If possible, your application should eliminate the need for multiple executions by caching information that is returned from catalog functions. For example, the application might call **SQLGetTypeInfo** once and cache the elements of the result set on which the application depends. Few applications use all elements of the result set that a catalog function generates, so the cache of information should not be difficult to maintain.

Supplying Non-Null Arguments to Catalog Functions

Because catalog functions are slow, your application should invoke them as efficiently as possible. Passing null arguments to catalog functions can result in the driver generating time-consuming queries and can increase network traffic with unwanted result-set information.

Rather than pass the smallest number of non-null arguments necessary for a catalog function to return success, always supply as many non-null arguments as possible to catalog functions. Any information that the application can send the driver when it calls a catalog function can result in improved performance and reliability.

Consider a call to **SQLTables** in which an application requests information about a table named **Customers**. This call is often coded similarly to the following example:

```
rc = SQLTables (NULL, NULL, NULL, NULL, “Customers”, SQL_NTS,
     NULL);
```

A driver might turn this **SQLTables** call into SQL statements, as in the following example:

```
SELECT...FROM SysTables WHERE TableName = ‘Customers’
    UNION ALL
SELECT...FROM SysViews WHERE ViewName = ‘Customers’
    UNION ALL
SELECT...FROM SysSynonyms WHERE SynName = ‘Customers’
    ORDER BY...
```

Suppose that the result set for this example contains three **Customers** tables: one table that the user owns, one table that sales owns, and one view that management creates. Although it might be obvious that the intent of the application is to use the one that the user owns, it might not be obvious to the user which table to choose. If the application specifies the *OwnerName* argument for the **SQLTables** call, reliability and performance improve, and only one table is returned. (Less network traffic is required to return only one result row, and the data source filters unwanted rows.)

To extend this example, if the application can provide a value for the *TableType* argument, the driver can optimize the SQL statement, as in the following command:

```
SELECT ... FROM SysTables WHERE TableName = ‘Customers’ AND
    Owner = ‘Beth’
```

Determining Table Characteristics

When an application calls **SQLColumns**, the database server must evaluate the query and form a result set to return to the client. When the application makes a dummy query with **SQLDescribeCol**, the database server does not execute the query; it only prepares it. Thus, no results are sent back to the client.

Consider an application that allows the user to choose which columns to select. The following pseudocode fragments and analysis illustrate the performance advantage of **SQLDescribeCol**.

Case 1: SQLColumns Method

The following example uses **SQLColumns**:

```
rc = SQLColumns (... “UnknownTable” ...);
// This call to SQLColumns will generate a query to the system
// catalogs... possibly a join which must be prepared,
// executed, and produce a result set
rc = SQLBindCol (...);
rc = SQLExtendedFetch (...);
// user must retrieve N rows from the server
// N = # result columns of UnknownTable
// result column information has now been obtained
```

Case 2: SQLDescribeCol Method

The following example uses **SQLDescribeCol**:

```
// prepare dummy query
rc = SQLPrepare (... “SELECT * from UnknownTable
    WHERE 1 = 0” ...);
// query is never executed on the server - only prepared
rc = SQLNumResultCols (...);
for (irow = 1; irow <= NumColumns; irow++)
     {
     rc = SQLDescribeCol (...)
     // + optional calls to SQLColAttributes
     }
// result column information has now been obtained
// Note we also know the column ordering within the table!
// This information cannot be
// assumed from the SQLColumns example.
```

Both cases send a query to the database server, but in Case 1, **SQLColumns** must evaluate the query and form a result set that it must send to the client. In Case 2, **SQLDescribeCol** gets the result-column information and the order of the columns within the table without returning the information to the client, thus improving performance.

Avoid using **SQLColumns** to determine characteristics of a table. Instead, use a dummy query with **SQLDescribeCol**.

Retrieving Data

To increase the performance of an application, limit the amount of data retrieved and reduce the call load.

Retrieving Long Data

Retrieving long data (SQL_LONGVARCHAR and SQL_LONGVARBINARY data) across the network is resource intensive and slow. Unless it is absolutely necessary, your application should avoid requesting long data. If the user wants to see long data items, the application can requery the database and specify only the long columns in the SELECT list.

In general, users do not want to see long data. Consequently, a default that does not retrieve long data or binary data is acceptable. If a user does want to see these result items, the application can requery the database, specifying only the long columns in the SELECT list. This method allows the average user to retrieve the result set without paying a high performance penalty for intense network traffic.

Although the optimal method is to exclude long data from the SELECT list, some applications do not formulate the select list before they send the query to the driver; that is, some applications use a statement such as the following one:

```
SELECT * FROM table_name...
```

If the SELECT list contains long data, some drivers *must* retrieve that data at fetch time even if the application does not bind the long data in the result set. If possible, the designer should attempt to implement a method that does not retrieve all columns of the table.

Using SQLSetStmtOption to Reduce the Size of Data Retrieved

When you retrieve long data, your result set can be huge. To reduce the size of the result set data to a manageable level, call **SQLSetStmtOption** with the SQL_MAX_LENGTH option.

The SQL_MAX_LENGTH option reduces network traffic and improves performance by allowing the driver to communicate to the database server that only *n* bytes of data are pertinent to the client. The database server responds by sending only the first *n* bytes of data for *all* result columns.

A common assumption among application developers is that if an application calls **SQLGetData** with a container of size *x*, then the driver retrieves only *x* bytes of information from the server. On the contrary, because an application can call **SQLGetData** multiple times for any one column, the INFORMIX-CLI driver retrieves long data in large chunks to optimize network use; it then returns the long data to the user upon request. The following code example illustrates the impact on performance:

```
char CaseContainer[1000];
...
rc = SQLExecDirect (hstmt, “SELECT CaseHistory FROM Cases
WHERE CaseNo = 71164”, SQL_NTS);
...
rc = SQLFetch (hstmt);
rc = SQLGetData (hstmt, 1, CaseContainer,(SWORD)
sizeof(CaseContainer), ...);
```

At this point, the driver is more likely to retrieve 64 kilobytes of information from the server instead of 1000 bytes. (In terms of network access, one 64 kilobytes retrieval is less expensive than sixty-four 1000-byte retrievals.) Unfortunately, the application might not call **SQLGetData** again; thus, the first and only retrieval of CaseHistory is slowed because 64 kilobytes of data has to be sent across the network.

In this example, only one row is returned. The performance impact is significant when, for example, 100 rows are returned in the result set. This example further illustrates how you can limit the size of the data retrieved with the SQL_MAX_LENGTH option of **SQLSetStmtOption** to improve performance.

Using SQLBindCol to Reduce the Call Load

Another way to improve performance is to retrieve data through bound columns. To reduce the ODBC call load, use **SQLBindCol** instead of **SQLGetData**.

The following pseudocode fragments and analysis illustrate the performance difference that can result from using **SQLBindCol**.

Case 1: SQLGetData Method

This example uses **SQLGetData**:

```
rc = SQLExecDirect (hstmt, “SELECT <20 columns> FROM Employees
    WHERE HireDate >= ?”, SQL_NTS);
do {
rc  = SQLFetch (hstmt);
// call SQLGetData 20 times
} while ((rc == SQL_SUCCESS) ||
    (rc == SQL_SUCCESS_WITH_INFO));
```

Case 2: SQLBindCol Method

This example uses **SQLBindCol**:

```
rc = SQLExecDirect (hstmt, “SELECT <20 columns> FROM Employees
    WHERE HireDate >= ?”, SQL_NTS);
// call SQLBindCol 20 times
do {
rc  = SQLFetch (hstmt);
} while ((rc == SQL_SUCCESS) ||
    (rc == SQL_SUCCESS_WITH_INFO));
```

Consider a situation in which the query returns 90 result rows. In Case 1, the number of ODBC calls made is greater than 1,890 (20 calls to **SQLGetData** x 90 result rows + 91 calls to **SQLFetch**). In Case 2, the number of ODBC calls made is reduced to about 110 (20 calls to **SQLBindCol** + 91 calls to **SQLFetch**). Clearly, the use of **SQLBindCol** can improve performance.

Another reason to use **SQLBindCol** is that many drivers optimize use of **SQLBindCol** by binding result information directly from the data source into the user’s buffer. That is, instead of the driver retrieving information into a container and then copying that information to the user’s buffer, the driver simply requests that the information from the database server be placed directly into the user’s buffer.

Using SQLExtendedFetch Versus SQLFetch

Use **SQLExtendedFetch** instead of **SQLFetch** to retrieve data. The ODBC call load decreases (resulting in better performance), and the code is less complex (resulting in more maintainable code).

Consider the preceding **SQLBindCol** example. The following code example uses **SQLExtendedFetch** instead of **SQLFetch**:

```
rc = SQLSetStmtOption (hstmt, SQL_ROWSET_SIZE, 100);
// use arrays of 100 elements
rc = SQLExecDirect (hstmt, “SELECT <20 columns> FROM
    Employees WHERE HireDate >= ?”, SQL_NTS);
// call SQLBindCol 1 time specifying row-wise binding
do {
rc = SQLExtendedFetch (hstmt, SQL_FETCH_NEXT, 0,
    &RowsFetched,RowStatus);
} while ((rc == SQL_SUCCESS) ||
    (rc == SQL_SUCCESS_WITH_INFO));
```

The number of ODBC calls that the application makes is reduced from 110 in Case 2 of the previous example to four calls (1 **SQLSet**St**mt**O**ption** + 1 **SQLExec**D**irect** + 1 **SQLBind**C**ol** + 1 **SQLExtended**F**etch**). The combined use of **SQLBind**C**ol** **SQLExtended**F**etch** reduces the initial call load of more than 1,890 ODBC calls (in “Case 1: SQLGetData Method” on page 10-9) to only four calls.

Executing Calls

Certain functions are more efficient than others at performing specialized tasks.

Using SQLPrepare and SQLExecute Versus SQLExecDirect

The combination of **SQLPrepare** and **SQLExecute** is optimized for multiple executions of a statement that most likely uses parameter markers. **SQLExecDirect** is optimized for a single execution of an SQL statement. Therefore, for better performance, use **SQLPrepare** and **SQLExecute** for queries that are executed more than once and **SQLExecDirect** for queries that are executed only once.

Unfortunately, more than 75 percent of all ODBC applications use **SQLPrepare** and **SQLExecute** exclusively. The following situation illustrates the pitfall of always coding **SQLPrepare** and **SQLExecute**.

Consider a driver that implements **SQLPrepare** by creating a stored procedure on the database server that contains the prepared statement. Creating a stored procedure has substantial overhead, but the driver assumes that the statement will be executed multiple times. In this case, although stored procedure creation is relatively expensive, execution is minimal because the query is parsed, and optimization paths are stored when the procedure is created. However, for such a driver, if the statement is executed only once, unneeded overhead results.

In general, applications that use **SQLPrepare** and **SQLExecute** for large, single-execution query batches almost certainly cause poor performance.

Similarly, applications that always use **SQLExecDirect** cannot perform as well as those that logically use a combination of **SQLPrepare, SQLExecute,** and **SQLExecDirect** sequences.

Updating Data

Use positioned updates and deletes and **SQLSpecialColumns** to improve the performance of your application.

Using Positioned Updates and Deletes

Although positioned updates do not apply to all types of applications, try to use positioned updates and deletes whenever possible. Positioned updates (with UPDATE WHERE CURRENT OF CURSOR) allow you to update data by positioning the database cursor to the row to be changed and signaling the driver to change the data. You are not forced to build a complex SQL statement; you simply supply the data to be changed.

Besides making the code more maintainable, positioned updates typically result in improved performance. Because the database server is already positioned on the row (for the SELECT statement currently in process), expensive operations to locate the row to be changed are unnecessary. If the row must be located, the database server typically has an internal pointer to the row available (for example, ROWID).

Determining the Optimal Set of Columns

Use **SQLSpecialColumns** to determine the optimal set of columns to use in the WHERE clause for updating data. Many times, pseudocolumns provide the fastest access to the data; you can determine these columns only by using **SQLSpecialColumns**.

Many applications cannot be designed to take advantage of positioned updates and deletes. These applications typically update data by forming a WHERE clause that consists of some subset of the column values that are returned in the result set. Some applications might formulate the WHERE clause by using all searchable result columns or by calling **SQLStatistics** to find columns that might be part of a unique index. These methods typically work but can result in fairly complex queries.

Consider the following example:

```
rc = SQLExecDirect (hstmt, “SELECT first_name, last_name, ssn,
     address, city, state, zip FROM emp”, SQL_NTS);
// fetchdata
:
:
rc = SQLExecDirect (hstmt, “UPDATE EMP SET ADDRESS = ?
     WHERE first_name = ? AND last_name = ? AND ssn = ?
     AND address = ? AND city = ? AND state = ? AND zip = ?”,
     SQL_NTS);
// fairly complex query
```

Applications should call **SQLSpecialColumns** / SQL_BEST_ROWID to retrieve the optimal set of columns (possibly a pseudocolumn) that identifies any given record. Many databases support special columns that are not explicitly user-defined in the table definition but are *hidden* columns of every table (for example, ROWID, TID, and so on). These pseudocolumns almost always provide the fastest access to the data because they typically are pointers to the exact location of the record. Because pseudocolumns are not part of the explicit table definition, they are not returned from **SQLColumns**. The only way to determine whether pseudocolumns exist is to call **SQLSpecialColumns**.

Consider the previous example, this time using **SQLSpecialColumns**:

```
:
:
rc = SQLSpecialColumns (hstmt, ..... ‘emp’, ...);
:
:
rc = SQLExecDirect (hstmt, “SELECT first_name, last_name, ssn,
     address, city, state, zip, ROWID FROM emp”, SQL_NTS);
// fetch data and probably “hide” ROWID from the user
:
:
rc = SQLExecDirect (hstmt, “UPDATE emp SET address = ? WHERE
     ROWID = ?”, SQL_NTS);
// fastest access to the data!
```

If your data source does not contain special pseudocolumns, the result set of **SQLSpecialColumns** consists of the columns of the optimal unique index on the specified table (if a unique index exists). Therefore, your application need not additionally call **SQLStatistics** to find the smallest unique index.

Committing Data

Committing data is extremely I/O-intensive and slow. By default, auto-commit is on when a driver connects to a data source. To improve performance, turn auto-commit off.

During a commit, the database server must flush back to disk every data page that contains updated or new data. The database server does not use a sequential write to perform this task. Instead, it uses a searched write to replace existing data that is already in the table. Thus, auto-commit mode is typically detrimental to performance because of the amount of I/O needed to commit *every* operation.

Chapter

Function Summary

11

This chapter summarizes the functions that INFORMIX-CLI-enabled applications and related software use:

- INFORMIX-CLI functions
- Setup shared-library functions
- Translation shared-library functions

INFORMIX-CLI Function Summary

The following tables list INFORMIX-CLI functions according to the type of tasks that the functions perform. The tables include function name, conformance designation, and a brief description of each function. For more information about conformance designations, see "API Conformance Levels" on page 3-3. For more information about the syntax and semantics for each function, see Chapter 12.

An application can call the **SQLGetInfo** function to get conformance information about a driver. To get information about support for a specific function in a driver, an application can call **SQLGetFunctions**.

Connecting to a Data Source

The following table describes functions that connect to a data source.

Function Name	ODBC Conformance	Purpose
SQLAllocEnv	Core	Obtains an environment handle. One environment handle is used for one or more connections.
SQLAllocConnect	Core	Obtains a connection handle.
SQLConnect	Core	Connects to a specific driver by data source name, user ID, and password.
SQLDriverConnect	Level 1	Connects to a specific driver by connection string or requests that INFORMIX-CLI display connection dialog boxes for the user.
SQLBrowseConnect	Level 2	Returns successive levels of connection attributes and valid attribute values. When a value is specified for each connection attribute, the function connects to the data source.

Retrieving Information About a Driver and Data Source

The following table describes functions that return information about a driver and data source.

Function Name	ODBC Conformance	Purpose
SQLDataSources	Level 2	Returns the list of available data sources.
SQLDrivers	Level 2	Returns the list of installed drivers and their attributes.
SQLGetInfo	Level 1	Returns information about a specific driver and data source.
SQLGetFunctions	Level 1	Returns supported driver functions.
SQLGetTypeInfo	Level 1	Returns information about supported data types.

Setting and Retrieving Driver Options

The following table describes functions that set and retrieve driver options.

Function Name	ODBC Conformance	Purpose
SQLSetConnectOption	Level 1	Sets a connection option.
SQLGetConnectOption	Level 1	Returns the value of a connection option.
SQLSetStmtOption	Level 1	Sets a statement option.
SQLGetStmtOption	Level 1	Returns the value of a statement option.

Preparing SQL Requests

The following table describes functions that prepare SQL requests.

Function Name	ODBC Conformance	Purpose
SQLAllocStmt	Core	Allocates a statement handle.
SQLPrepare	Core	Prepares an SQL statement for later execution.
SQLBindParameter	Level 1	Assigns storage for a parameter in an SQL statement.
SQLGetCursorName	Core	Returns the cursor name associated with a statement handle.
SQLSetCursorName	Core	Specifies a cursor name.
SQLSetScrollOptions	Level 2	Sets options that control cursor behavior.

Submitting SQL Requests

The following table describes functions that submit SQL requests.

Function Name	ODBC Conformance	Purpose
SQLExecute	Core	Executes a prepared statement.
SQLExecDirect	Core	Executes a statement.
SQLNativeSql	Level 2	Returns the text of an SQL statement as translated by the driver.

(1 of 2)

Function Name	ODBC Conformance	Purpose
SQLNumParams	Level 2	Returns the number of parameters in a statement.
SQLParamData	Level 1	Used with **SQLPutData** to supply parameter data at execution time. (Useful for long data values.)
SQLPutData	Level 1	Sends part or all of a data value for a parameter. (Useful for long data values.)

(2 of 2)

Retrieving Results and Information About Results

The following table describes functions that retrieve results and information about results.

Function Name	ODBC Conformance	Purpose
SQLRowCount	Core	Returns the number of rows affected by an insert, update, or delete request.
SQLNumResultCols	Core	Returns the number of columns in the result set.
SQLDescribeCol	Core	Describes a column in the result set.
SQLColAttributes	Core	Describes attributes of a column in the result set.
SQLBindCol	Core	Assigns storage for a result column and specifies the data type.
SQLFetch	Core	Returns a result row.
SQLExtendedFetch	Level 2	Returns multiple result rows.

(1 of 2)

Function Name	ODBC Conformance	Purpose
SQLGetData	Level 1	Returns part or all of one column of one row of a result set. (Useful for long data values.)
SQLMoreResults	Level 2	Determines whether more result sets are available and, if so, initializes processing for the next result set.
SQLError	Core	Returns additional error or status information.

(2 of 2)

Retrieving Information About Data-Source System Tables

The following table describes catalog functions that return information about data-source system tables.

Function Name	ODBC Conformance	Purpose
SQLColumnPrivileges	Level 2	Returns a list of columns and associated privileges for one or more tables.
SQLColumns	Level 1	Returns the list of column names in specified tables.
SQLForeignKeys	Level 2	Returns a list of column name or names that make up foreign keys, if they exist for a specific table.
SQLPrimaryKeys	Level 2	Returns the list of column name or names that make up the primary key for a table.
SQLProcedureColumns	Level 2	Returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures.

(1 of 2)

Function Name	ODBC Conformance	Purpose
SQLProcedures	Level 2	Returns a list of procedure names stored in a specific data source.
SQLSpecialColumns	Level 1	Returns information about the optimal set of columns that uniquely identifies a row in a specified table, or the columns that are automatically updated when a transaction updates any value in the row.
SQLStatistics	Level 1	Returns statistics about a single table and the list of indexes associated with the table.
SQLTablePrivileges	Level 2	Returns a list of tables and the privileges associated with each table.
SQLTables	Level 1	Returns the list of table names stored in a specific data source.

(2 of 2)

Terminating a Statement

The following table describes functions that terminate a statement.

Function Name	ODBC Conformance	Purpose
SQLFreeStmt	Core	Ends statement processing and closes the associated cursor, discards pending results, and, optionally, frees all resources associated with the statement handle.
SQLCancel	Core	Cancels an SQL statement.
SQLTransact	Core	Commits or rolls back a transaction.

Terminating a Connection

The following table describes functions that terminate a connection.

Function Name	ODBC Conformance	Purpose
SQLDisconnect	Core	Closes the connection.
SQLFreeConnect	Core	Releases the connection handle.
SQLFreeEnv	Core	Releases the environment handle.
SQLMoreResults	Level 2	Determines whether more results are available on a *hstmt*.

Setup Shared-Library Function Summary

The following table describes setup shared-library functions. For more information about the syntax and semantics for each function, see Chapter 13, "Setup Shared Library Function Reference."

Task	Function Name	Purpose
Setting up data sources and translators	**ConfigDSN**	Adds, modifies, or deletes a data source.
	ConfigTranslator	Returns a default translation option.

Translation Shared-Library Function Summary

The following table describes translation shared-library functions.

Task	Function name	Purpose
Translating data	**SQLDriverToDataSource**	Translates all data that flows from the driver to the data source.
	SQLDataSourceToDriver	Translates all data that flows from the data source to the driver.

Chapter

INFORMIX-CLI Function Reference

12

This chapter describes each INFORMIX-CLI function. The functions are listed alphabetically. Each function is defined as a programming-language function. The function descriptions include the following aspects:

- Conformance level
- Purpose
- Syntax
- Return codes
- Diagnostics
- Usage
- Code example (optional)
- Related functions

Error handling is described under the **SQLError** function description. The text associated with SQLSTATE values is included to provide a description of the condition, but it is not intended to prescribe specific text.

Arguments

All INFORMIX-CLI function arguments use a naming convention of the following form:

```
[[prefix]...]tag[qualifier][suffix]
```

Optional elements appear in brackets ([ ]) and use the following prefixes.

Prefix	Description
c	Count of
h	Handle of
i	Index of
p	Pointer to
rg	Range (array) of

The following tags are used.

Tag	Description
b	Byte
col	Column (of a result set)
dbc	Database connection
env	Environment
f	Flag (enumerated type)
par	Parameter (of an SQL statement)
row	Row (of a result set)
stmt	Statement
sz	Character string (array of characters, terminated by zero)
v	Value of unspecified type

Prefixes and tags combine to correspond roughly to the typedefs for the standard C data types listed below. Flags (f) and byte counts (cb) do not distinguish among SWORD, UWORD, SDWORD, and UDWORD.

Combined	Prefix	Tag	Typedefs for Standard C Data Types	Description
cb	c	b	SWORD, SDWORD, UDWORD	Count of bytes
crow	c	row	SDWORD, UDWORD, UWORD	Count of rows
f	–	f	SWORD, UWORD	Flag
hdbc	h	dbc	HDBC	Connection handle
henv	h	env	HENV	Environment handle
hstmt	h	stmt	HSTMT	Statement handle
hwnd	h	wnd	HWND	Widget
ib	i	b	SWORD	Byte index
icol	i	col	UWORD	Column index
ipar	i	par	UWORD	Parameter index
irow	i	row	SDWORD, UWORD	Row index
pcb	pc	b	SWORD FAR *, SDWORD FAR *, UDWORD FAR *	Pointer to byte count
pccol	pc	col	SWORD FAR *	Pointer to column count
pcpar	pc	par	SWORD FAR *	Pointer to parameter count
pcrow	pc	row	SDWORD FAR *, UDWORD FAR *	Pointer to row count
pf	p	f	SWORD, SDWORD, UWORD	Pointer to flag
phdbc	ph	dbc	HDBC FAR *	Pointer to connection handle

(1 of 2)

Combined	Prefix	Tag	Typedefs for Standard C Data Types	Description
phenv	ph	env	HENV FAR *	Pointer to environment handle
phstmt	ph	stmt	HSTMT FAR *	Pointer to statement handle
pib	pi	b	SWORD FAR *	Pointer to byte index
pirow	pi	row	UDWORD FAR *	Pointer to row index
prgb	prg	b	PTR FAR *	Pointer to range (array) of bytes
pv	p	v	PTR	Pointer to value of unspecified type
rgb	rg	b	PTR	Range (array) of bytes
rgf	rg	f	UWORD FAR *	Range (array) of flags
sz	–	sz	UCHAR FAR *	String, zero terminated
v	–	v	UDWORD	Value of unspecified type

(2 of 2)

Qualifiers are used to distinguish specific variables of the same typedef. Qualifiers consist of the concatenation of one or more capitalized English words or abbreviations.

INFORMIX-CLI defines one value for the suffix *Max*, which denotes that the variable represents the largest value of its type for a given situation.

For example, the argument *cbErrorMsgMax* contains the largest possible byte count for an error message; in this case, the argument corresponds to the size in bytes of the argument *szErrorMsg*, a character string buffer. The argument *pcbErrorMsg* is a pointer to the count of bytes available to return in the argument *szErrorMsg*, not including the null termination character.

Important: *Because characters are signed in many UNIX implementations and string arguments are unsigned in INFORMIX-CLI functions, applications that pass string objects to INFORMIX-CLI functions without casting them receive compiler warnings.*

INFORMIX-CLI Include Files

The files **sql.h** and **sqlext.h** contain function prototypes for all the INFORMIX-CLI functions. They also contain all type definitions and **#define** constants.

Diagnostics

The diagnostics provided with each function list SQLSTATE values that the function might return. INFORMIX-CLI can return additional SQLSTATE values that arise from implementation-specific situations.

The character string value returned for an SQLSTATE consists of a two-character class value followed by a three-character subclass value. A class value of `01` indicates a warning and is accompanied by a return code of SQL_SUCCESS_WITH_INFO. Class values other than `01`, except for the class `IM`, indicate an error and are accompanied by a return code of SQL_ERROR. The class `IM` signifies warnings and errors that derive from the implementation of INFORMIX-CLI. The subclass value `000` in any class is for implementation-defined conditions within the given class. ANSI SQL-92 defines the assignment of class and subclass values.

Tables and Views

In INFORMIX-CLI functions, tables and views are interchangeable. The term *table* is used for tables and views, except where view is used explicitly.

Catalog Functions

The functions listed in "Retrieving Information About Data-Source System Tables" on page 11-8 are catalog functions that return information about a data-source catalog.

Search Pattern Arguments

Each catalog function returns information in the form of a result set. The information that a function returns can be constrained by a search pattern passed as an argument to that function. These search patterns can contain the underscore (_), the percent symbol (%), and a driver-defined escape character, as follows:

- The underscore represents any single character.
- The percent symbol represents any sequence of zero or more characters.
- The escape character, backslash (\), permits the underscore and percent metacharacters to be used as literal characters in search patterns. To use a metacharacter as a literal character in the search pattern, precede it with the escape character. To use the escape character as a literal character in the search pattern, include it twice.
- All other characters represent themselves.

For example, if the search pattern for a table name is %A%, the function returns all tables with names that contain the character *A*. If the search pattern for a table name is B__ (B followed by two underscores), the function returns all tables with names that are three characters long and start with the character *B*. If the search pattern for a table name is %, the function returns all tables.

If the search pattern for a table name is ABC\%, the function returns the table named `ABC%`. If the search pattern for a table name is \\%, the function returns all tables with names that start with a backslash. Failing to precede a metacharacter used as a literal with the escape character might return more results than expected. For example, if **SQLTables** returns a table identifier **MY_TABLE** and if an application wants to retrieve a list of columns for **MY_TABLE** using **SQLColumns, SQLColumns** returns all the tables that match **MY_TABLE**, such as **MY_TABLE**, **MY1TABLE**, **MY2TABLE**, and so on, unless the escape character precedes the underscore.

Important: *A zero-length search pattern matches the empty string. A search-pattern argument that is a null pointer means the search is not constrained for the argument. A null pointer and a search string of "%" should return the same values.*

Core

SQLAllocConnect

SQLAllocConnect allocates memory for a connection handle within the environment that *henv* identifies.

Syntax

```
RETCODE SQLAllocConnect(henv, phdbc)
```

The **SQLAllocConnect** function accepts the following arguments.

Typedef	Argument	Use	Description
HENV	*henv*	Input	Environment handle
HDBC FAR *	*phdbc*	Output	Pointer to storage for the connection handle

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

If **SQLAllocConnect** returns SQL_ERROR, it sets the *hdbc* referenced by *phdbc* to SQL_NULL_HDBC. To obtain additional information, the application can call **SQLError** with the specified *henv* and with *hdbc* and *hstmt* set to SQL_NULL_HDBC and SQL_NULL_HSTMT, respectively.

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	(DM) The driver manager could not allocate memory for the connection handle. The driver could not allocate memory for the connection handle.
S1009	Invalid argument value	(DM) The argument *phdbc* was a null pointer.

Usage

A connection handle references information such as the valid statement handles on the connection and whether a transaction is currently open. To request a connection handle, an application passes the address of an *hdbc* to **SQLAllocConnect**. The driver allocates memory for the connection information and stores the value of the associated handle in the *hdbc*. On operating systems that support multiple threads, applications can use the same *hdbc* on different threads. The application passes the *hdbc* value in all subsequent calls that require an *hdbc*.

If your INFORMIX-CLI architecture includes a driver manager, the driver manager processes the **SQLAllocConnect** function and calls the driver **SQLAllocConnect** function when the application calls **SQLConnect**, **SQLBrowseConnect**, or **SQLDriverConnect**. For more information, see "Usage" on page 12-79.

If the application calls **SQLAllocConnect** with a pointer to a valid *hdbc*, the driver overwrites the *hdbc* without regard to its previous contents.

Code Example

See **SQLBrowseConnect** and **SQLConnect**.

Related Functions

For Information About	See
Connecting to a data source	**SQLConnect**
Freeing a connection handle	**SQLFreeConnect**

♦

Core

SQLAllocEnv

SQLAllocEnv allocates memory for an environment handle and initializes the INFORMIX-CLI call level interface for application use. An application must call **SQLAllocEnv** before it calls any other INFORMIX-CLI function.

Syntax

```
RETCODE SQLAllocEnv(phenv)
```

The **SQLAllocEnv** function accepts the following argument.

Typedef	Argument	Use	Description
HENV FAR *	*phenv*	Output	Pointer to storage for the environment handle

Return Codes

SQL_SUCCESS or SQL_ERROR

If **SQLAllocEnv** returns SQL_ERROR, it sets the *henv* that *phenv* references to SQL_NULL_HENV. In this case, the application can assume that the error was a memory-allocation error.

Diagnostics

A driver cannot return SQLSTATE values directly after the call to **SQLAllocEnv** because no valid handle exists with which to call **SQLError**.

Two levels of **SQLAllocEnv** functions exist, one within the driver manager (if you are using one) and one within the driver. The driver manager does not call the driver-level function until the application calls **SQLConnect**, **SQLBrowseConnect**, or **SQLDriverConnect**. If an error occurs in the driver-level **SQLAllocEnv** function, the driver manager-level **SQLConnect**, **SQLBrowseConnect**, or **SQLDriverConnect** function returns SQL_ERROR. A subsequent call to **SQLError** with *henv*, SQL_NULL_HDBC, and SQL_NULL_HSTMT returns SQLSTATE IM004 (the driver **SQLAllocEnv** function failed), followed by one of the following errors from the driver:

- SQLSTATE S1000 (General error)
- An INFORMIX-CLI SQLSTATE value, which ranges from S1000 to S19ZZ. For example, SQLSTATE S1001 (Memory-allocation failure) indicates that the call from the driver manager to the driver-level **SQLAllocEnv** returned SQL_ERROR, and the *henv* from the driver manager was set to SQL_NULL_HENV.

For additional information about the flow of function calls between the driver manager and a driver, see "Usage" on page 12-79.

Usage

An environment handle references global information such as valid connection handles and active connection handles. To request an environment handle, an application passes the address of an *henv* to **SQLAllocEnv**. The driver allocates memory for the environment information and stores the value of the associated handle in the *henv*. On operating systems that support multiple threads, applications can use the same *hdbc* on different threads. The application passes the *henv* value in all subsequent calls that require an *henv*.

More than one *henv* should never be allocated at one time, and the application should not call **SQLAllocEnv** when a current valid *henv* exists. If the application calls **SQLAllocEnv** with a pointer to a valid *henv*, the driver overwrites the *henv* without regard to its previous contents.

When INFORMIX-CLI processes the **SQLAllocEnv** function, it checks the **Trace** value in the initialization files. If the value is 1, INFORMIX-CLI enables tracing for all applications.

Code Example

See **SQLBrowseConnect** and **SQLConnect**.

Related Functions

For Information About	See
Allocating a connection handle	**SQLAllocConnect**
Connecting to a data source	**SQLConnect**
Freeing an environment handle	**SQLFreeEnv**

♦

Core

SQLAllocStmt

SQLAllocStmt allocates memory for a statement handle and associates the statement handle with the connection that *hdbc* specifies.

An application must call **SQLAllocStmt** before it submits SQL statements.

Syntax

```
RETCODE SQLAllocStmt(hdbc, phstmt)
```

The **SQLAllocStmt** function accepts the following arguments.

Typedef	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle
HSTMT FAR *	*phstmt*	Output	Pointer to storage for the statement handle

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, or SQL_ERROR

If **SQLAllocStmt** returns SQL_ERROR, it sets the *hstmt* that *phstmt* references to SQL_NULL_HSTMT. The application can then obtain additional information by calling **SQLError** with the *hdbc* and SQL_NULL_HSTMT.

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	Informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	(DM) The connection that the *hdbc* argument specifies was not open. The connection process must be completed successfully (and the connection must be open) for the driver to allocate an *hstmt*.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	(DM) The driver manager could not allocate memory for the statement handle. The driver could not allocate memory for the statement handle.
S1009	Invalid argument value	(DM) The argument *phstmt* was a null pointer.

Usage

A statement handle references statement information, such as network information, SQLSTATE values and error messages, cursor name, several result-set columns, and status information for SQL statement processing.

To request a statement handle, an application connects to a data source and then passes the address of an *hstmt* to **SQLAllocStmt**. The driver allocates memory for the statement information and stores the value of the associated handle in the *hstmt*. On operating systems that support multiple threads, applications can use the same *hdbc* on different threads. The application passes the *hstmt* value in all subsequent calls that require an *hstmt*.

If the application calls **SQLAllocStmt** with a pointer to a valid *hstmt*, the driver overwrites the *hstmt* without regard to its previous contents.

Code Example

See **SQLBrowseConnect**, **SQLConnect**, and **SQLSetCursorName**.

Related Functions

For Information About	See
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Freeing a statement handle	**SQLFreeStmt**
Preparing a statement for execution	**SQLPrepare**

♦

Core

SQLBindCol

SQLBindCol assigns the storage and INFORMIX-CLI C data type for a column in a result set, as follows:

- A storage buffer that receives the contents of a column of data
- The length of the storage buffer
- A storage location that receives the actual length of the column of data returned by the fetch operation
- Data type conversion from the Informix SQL data type to the INFORMIX-CLI C data type

Syntax

```
RETCODE SQLBindCol(hstmt, icol, fCType, rgbValue, cbValueMax,
pcbValue)
```

The **SQLBindCol** function accepts the following arguments.

Typedef	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UWORD	*icol*	Input	Column number of result data, ordered sequentially left to right, starting at 1.
SWORD	*fCType*	Input	The INFORMIX-CLI C data type for the result data. The possible values are any of the *fCType* values listed in Appendix B, “Data Types,” as well as SQL_C_DEFAULT. For more information about data types and conversions, see Appendix B.
PTR	*rgbValue*	Input	Pointer to storage for the data. If *rgbValue* is a null pointer, the driver unbinds the column. (To unbind all columns, an application calls **SQLFreeStmt** with the SQL_UNBIND option.)

(1 of 2)

Typedef	Argument	Use	Description
SDWORD	*cbValueMax*	Input	Maximum length of the *rgbValue* buffer. For character data, *rgbValue* must also include space for the null-termination byte. For more information about length, see "Precision, Scale, Length, and Display Size" on page B-5.
SDWORD FAR *	*pcbValue*	Input	SQL_NULL_DATA or the number of bytes (excluding the null-termination byte for character data) available to return in *rgbValue* prior to calling **SQLExtendedFetch** or **SQLFetch**, or SQL_NO_TOTAL if the function cannot determine the number of available bytes. For character data, if the number of bytes available to return is SQL_NO_TOTAL or is greater than or equal to *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax* – 1 bytes and is null-terminated by the driver. For binary data, if the number of bytes available to return is SQL_NO_TOTAL or is greater than *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax* bytes. For all other types of data, the value of *cbValueMax* is ignored, and the driver assumes the size of *rgbValue* is the size of the INFORMIX-CLI C data type specified with *fCType*. For more information about the value returned in *pcbValue* for each *fCType*, see "Converting Data from SQL to C" on page B-19.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support executing or completing the function.
S1002	Invalid column number	The value specified for the argument *icol* exceeded the maximum number of columns that the data source supports.
S1003	Program type out of range	(DM) The argument *fCType* was not a valid INFORMIX-CLI C data type or SQL_C_DEFAULT.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value specified for the argument *cbValueMax* was less than 0.
S1C00	Driver not capable	The driver does not support the INFORMIX-CLI C data type specified in the argument *fCType*. The argument *icol* was 0, and the driver does not support bookmarks.

Usage

The INFORMIX-CLI interface provides the following ways to retrieve a column of data:

- **SQLBindCol** assigns the storage location for a column of data before the data is retrieved. When **SQLFetch** or **SQLExtendedFetch** is called, the driver places the data for all bound columns in the assigned locations.
- **SQLGetData** (an extended function) assigns a storage location for a column of data after **SQLFetch** or **SQLExtendedFetch** is called. It also places the data for the requested column in the assigned location. Because it can retrieve data from a column in parts, **SQLGetData** can retrieve long data values.

An application might choose to bind every column with **SQLBindCol**, to retrieve data only (and not bind) with **SQLGetData**, or to use a combination. However, unless the driver provides extended functionality, **SQLGetData** can be used only to retrieve data from columns that occur after the last bound column.

An application calls **SQLBindCol** to pass the pointer to the storage buffer for a column of data to the driver and to specify how or if to convert the data. The application must allocate enough storage for the data. If the buffer contains variable-length data, the application must allocate as much storage as the maximum length of the bound column, or the data might be truncated. For information about converting data types, see "Converting Data from SQL to C" on page B-19.

At fetch time, the driver processes the data for each bound column according to the arguments specified in **SQLBindCol**. First, it converts the data according to the argument *fCType*. Next, it fills the buffer to which *rgbValue* points. Finally, it stores the available number of bytes in *pcbValue*; the value stored in *pcbValue* is the number of bytes available before the driver calls **SQLFetch** or **SQLExtendedFetch**.

Before the driver calls **SQLFetch** or **SQLExtendedFetch,** it returns a value, according to the following conditions:

- If SQL_MAX_LENGTH has been specified with **SQLSetStmtOption** and the available number of bytes is greater than SQL_MAX_LENGTH, the driver stores SQL_MAX_LENGTH in *pcbValue*.
- If the data is truncated because of SQL_MAX_LENGTH, but the user's buffer is large enough for SQL_MAX_LENGTH bytes of data, the driver returns SQL_SUCCESS.

Important: *The SQL_MAX_LENGTH statement option is intended to reduce network traffic. To guarantee that data is truncated, an application should allocate a buffer of the desired size and specify this size in the cbValueMax argument.*

- If the user's buffer causes the truncation, the driver returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01004 (Data truncated) for the fetch function.
- If the data value for a column is NULL, the driver sets *pcbValue* to SQL_NULL_DATA.
- If the number of bytes available to return cannot be determined in advance, the driver sets *pcbValue* to SQL_NO_TOTAL.

When an application uses **SQLExtendedFetch** to retrieve more than one row of data, it needs to call **SQLBindCol** only once for each column of the result set (in the same way that it binds a column in order to retrieve a single row of data with **SQLFetch**). The **SQLExtendedFetch** function coordinates placing each row of data into subsequent locations in the row set buffers. For additional information about binding row set buffers, see "SQLExtendedFetch" on page 12-123.

An application can call **SQLBindCol** to bind a column to a new storage location, regardless of whether data has already been fetched. The new binding replaces the old binding. The new binding does not apply to data already fetched; it takes effect the next time **SQLFetch** or **SQLExtendedFetch** is called.

To unbind a single-bound column, an application calls **SQLBindCol** and specifies a null pointer for *rgbValue*; if *rgbValue* is a null pointer and the column is not bound, **SQLBindCol** returns SQL_SUCCESS. To unbind all bound columns, an application calls **SQLFreeStmt** with the SQL_UNBIND option.

Code Example

In the following example, an application executes a SELECT statement to return a result set of the employees' names, ages, and birthdays, which is sorted by birthday. It then calls **SQLBindCol** to bind the columns of data to local storage locations. Finally, the application fetches each row of data with **SQLFetch** and prints the name, age, and birthday of each employee.

For more code examples, see **SQLColumns** and **SQLExtendedFetch**.

```
#define NAME_LEN 30
#define BDAY_LEN 11

UCHAR  szName[NAME_LEN], szBirthday[BDAY_LEN];
SWORD  sAge;
SDWORD cbName, cbAge, cbBirthday;

retcode = SQLExecDirect(hstmt,
       "SELECT NAME, AGE, BIRTHDAY FROM EMPLOYEE ORDER BY 3, 2, 1",
       SQL_NTS);

if (retcode = = SQL_SUCCESS) {

    /* Bind columns 1, 2, and 3 */

    SQLBindCol(hstmt, 1, SQL_C_CHAR, szName, NAME_LEN, &cbName);
    SQLBindCol(hstmt, 2, SQL_C_SSHORT, &sAge, 0, &cbAge);
    SQLBindCol(hstmt, 3, SQL_C_CHAR, szBirthday, BDAY_LEN, &cbBirthday);

    /* Fetch and print each row of data.  On */
    /* an error, display a message and exit. */

    while (TRUE) {
        retcode = SQLFetch(hstmt);
        if (retcode = = SQL_ERROR || retcode = = SQL_SUCCESS_WITH_INFO) {
            show_error();
        }
        if (retcode = = SQL_SUCCESS || retcode = = SQL_SUCCESS_WITH_INFO){
            fprintf(out, "%-*s %-2d %*s", NAME_LEN-1, szName,

                    sAge, BDAY_LEN-1, szBirthday);
        } else {
            break;
        }
    }
}
```

Related Functions

For Information About	See
Returning information about a column in a result set	**SQLDescribeCol**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Freeing a statement handle	**SQLFreeStmt**
Fetching part or all of a column of data	**SQLGetData**
Returning the number of result set columns	**SQLNumResultCols**

♦

Level 1

SQLBindParameter

SQLBindParameter binds a buffer to a parameter marker in an SQL statement.

Syntax

```
RETCODE SQLBIND (hstmt, ipar, fParamType, fCType, fSqlType,
cbColDef, ibScale, rgbValue, cbValueMax, pcbValue)
```

The **SQLBindParameter** function accepts the following arguments.

Typedef	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle
UWORD	*ipar*	Input	Parameter number, ordered sequentially left to right, starting at 1
SWORD	*fParamType*	Input	The type of the parameter. For more information, see "fParamType Argument" on page 12-29.
SWORD	*fCType*	Input	The INFORMIX-CLI C data type of the result data. The possible values are any of the *fCType* values listed in Appendix B, "Data Types," as well as SQL_C_DEFAULT. For more information about data types and conversions, see Appendix B.
SWORD	*fSqlType*	Input	The INFORMIX-CLI SQL data type of the parameter. The possible values are any of the *fSqlType* values listed in Appendix B. For more information about data types and conversions, see Appendix B.
UDWORD	*cbColDef*	Input	The precision of the column or expression of the corresponding parameter marker. For more information, see "cbColDef Argument" on page 12-30.

(1 of 2)

Typedef	Argument	Use	Description
SWORD	*ibScale*	Input	The scale of the column or expression of the corresponding parameter marker. For further information concerning scale, see "Precision, Scale, Length, and Display Size" on page B-5.
PTR	*rgbValue*	Input/ Output	A pointer to a buffer for the data of the parameter. For more information, see "rgbValue Argument" on page 12-31.
SDWORD	*cbValueMax*	Input	Maximum length of the *rgbValue* buffer. For more information, see "cbValueMax Argument" on page 12-31.
SDWORD FAR *	*pcbValue*	Input/ Output	A pointer to a buffer for the length of the parameter. For more information, see "pcbValue Argument" on page 12-32.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When **SQLBindParameter** returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, an associated SQLSTATE value can be obtained by calling **SQLError**. The following table lists the SQLSTATE values commonly returned by **SQLBindParameter** and explains each value in the context of this function; the notation (DM) precedes the description of each SQLSTATE returned by the driver manager. The return code associated with each SQLSTATE value is SQL_ERROR unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
07006	Restricted data type attribute violation	The data value associated with the *fCType* argument cannot be converted to the INFORMIX-CLI SQL data type identified by the *fSqlType* argument.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support executing or completing the function.
S1003	Program type out of range	(DM) *fCType* was invalid.
S1004	SQL data type out of range	(DM) *fSqlType* was invalid.
S1009	Invalid argument value	(DM) The argument *rgbValue* was a null pointer, the argument *pcbValue* was a null pointer, and the argument *fParamType* was not SQL_PARAM_OUTPUT.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

(1 of 2)

SQLSTATE	Error	Description
S1090	Invalid string or buffer length	(DM) The value specified for the argument *cbValueMax* was less than 0.
S1093	Invalid parameter number	(DM) The value specified for the argument *ipar* was less than 1. The value specified for the argument *ipar* was greater than the maximum number of parameters supported by the data source.
S1094	Invalid scale value	*ibScale* was invalid for *fSqlType*.
S1104	Invalid precision value	*cbColDef* was invalid for *fSqlType*.
S1105	Invalid parameter type	(DM) *fParamType* was invalid. For more information, see "fParamType Argument" on page 12-29. *fParamType* was SQL_PARAM_OUTPUT and the parameter did not mark a return value from a procedure or a procedure parameter. *fParamType* was SQL_PARAM_INPUT and the parameter marked the return value from a procedure.
S1C00	Driver not capable	The driver or data source does not support the conversion specified by the combination of *fCType* and *fSqlType*. The driver or data source does not support the value specified for *fSqlType*.

(2 of 2)

Usage

An application calls **SQLBindParameter** to bind each parameter marker in an SQL statement. Bindings are effective until the application calls **SQLBindParameter** again or until the application calls **SQLFreeStmt** with the SQL_DROP or SQL_RESET_PARAMS option.

fParamType Argument

The *fParamType* argument specifies the parameter type. All parameters in SQL statements that do not call procedures, such as INSERT statements, are input parameters. Parameters in procedure calls can be input, input/output, or output parameters. (An application calls **SQLProcedureColumns** to determine the type of a parameter in a procedure call; parameters in procedure calls whose type cannot be determined are assumed to be input parameters.)

The *fParamType* argument is one of the following values:

- SQL_PARAM_INPUT

  This parameter marks a parameter in an SQL statement that does not call a procedure, such as an INSERT statement, or it marks an input parameter in a procedure; these are collectively known as *input parameters*. For example, the parameters in INSERT INTO Employee VALUES (?, ?, ?) and {call AddEmp(?, ?, ?)} are input parameters.

  When the statement executes, the driver sends data for the parameter to the data source; the *rgbValue* buffer must contain a valid input value or the *pcbValue* buffer must contain SQL_NULL_DATA, SQL_DATA_AT_EXEC, or the result of the SQL_LEN_DATA_AT_EXEC macro.

  If an application cannot determine the type of a parameter in a procedure call, it sets *fParamType* to SQL_PARAM_INPUT; if the data source returns a value for the parameter, the driver discards it.

- SQL_PARAM_INPUT_OUTPUT

  The parameter marks an input/output parameter in a procedure. For example, the parameter in {call GetEmpDept(?)} is an input/output parameter that accepts the name of an employee and returns the department name of the employee.

When the statement executes, the driver sends data for the parameter to the data source; the *rgbValue* buffer must contain a valid input value or the *pcbValue* buffer must contain SQL_NULL_DATA, SQL_DATA_AT_EXEC, or the result of the SQL_LEN_DATA_AT_EXEC macro. After the statement executes, the driver returns data for the parameter to the application; if the data source does not return a value for an input/output parameter, the driver sets the *pcbValue* buffer to SQL_NULL_DATA.

If an application calls **SQLSetParam**, the driver manager converts this to a call to **SQLBindParameter** in which the *fParamType* argument is set to SQL_PARAM_INPUT_OUTPUT.

- SQL_PARAM_OUTPUT

  The parameter marks the return value of a procedure or an output parameter in a procedure; these are collectively known as *output parameters*. For example, the parameter in {?=call GetNextEmpID} is an output parameter that returns the next employee ID.

  After the statement executes, the driver returns data for the parameter to the application, unless the *rgbValue* and *pcbValue* arguments are both null pointers, in which case the driver discards the output value. If the data source does not return a value for an output parameter, the driver sets the *pcbValue* buffer to SQL_NULL_DATA.

cbColDef Argument

The *cbColDef* argument specifies the precision of the column or expression that corresponds to the parameter marker, unless all of the following conditions are true:

- An application calls **SQLSetParam**. The driver manager converts these calls to **SQLBindParameter**.
- The *fSqlType* argument is SQL_LONGVARBINARY or SQL_LONGVARCHAR.
- The data for the parameter is sent with **SQLPutData**.

  In this case, the *cbColDef* argument contains the total number of bytes that are sent for the parameter. For more information, see "Passing Parameter Values" on page 12-33.

rgbValue Argument

The *rgbValue* argument points to a buffer that, when **SQLExecute** or **SQLExecDirect** is called, contains the actual data for the parameter. The data must be in the form specified by *fCType*.

If *pcbValue* is the result of the SQL_LEN_DATA_AT_EXEC(*length*) macro or SQL_DATA_AT_EXEC, then *rgbValue* is an application-defined 32-bit value that is associated with the parameter. It is returned to the application through **SQLParamData**. For example, *rgbValue* might be a token such as a parameter number, a pointer to data, or a pointer to a structure that the application used to bind input parameters. If the parameter is an input / output parameter, *rgbValue* must be a pointer to a buffer where the output value is stored.

If the *fParamType* argument is SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT, *rgbValue* points to a buffer in which the driver returns the output value. If the procedure returns one or more result sets, the *rgbValue* buffer is not guaranteed to be set until all results are fetched. If *fParamType* is SQL_PARAM_OUTPUT and *rgbValue* and *pcbValue* are both null pointers, the driver discards the output value.

cbValueMax Argument

For character and binary C data, the *cbValueMax* argument specifies the length of the *rgbValue* buffer, if it is a single element. If the application specifies multiple values, *cbValueMax* determines the location of values in the *rgbValue* array, both on input and on output. For input / output and output parameters, it determines whether to truncate character and binary C data on output.

For character C data, if the number of bytes available to return is greater than or equal to *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax* – 1 bytes and is null-terminated by the driver.

For binary C data, if the number of bytes available to return is greater than *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax* bytes.

For all other types of C data, the *cbValueMax* argument is ignored. The length of the *rgbValue* buffer (if it is a single element) is assumed to be the length of the INFORMIX-CLI C data type.

pcbValue Argument

The *pcbValue* argument points to a buffer that, when **SQLExecute** or **SQLExecDirect** is called, contains one of the values in the following table.

Value	Description
The length of the parameter value stored in *rgbValue*	This value is ignored except for character or binary C data.
SQL_NTS	The parameter value is a null-terminated string.
SQL_NULL_DATA	The parameter value is NULL.
SQL_DEFAULT_PARAM	The procedure is to use the default value of a parameter rather than a value retrieved from the application. This value is valid only in a procedure call and then only if the *fParamType* argument is SQL_PARAM_INPUT or SQL_PARAM_INPUT_OUTPUT. When *pcbValue* is SQL_DEFAULT_PARAM, the corresponding parameter can only be a parameter for an ODBC canonical procedure invocation. When *pcbValue* is SQL_DEFAULT_PARAM, the *fCType*, *fSqlType*, *cbColDef*, *ibScale*, *cbValueMax* and *rgbValue* arguments are ignored for input parameters and are used only to define the output parameter value for input/output parameters. This value was introduced in ODBC 2.0.
The result of the SQL_LEN_DATA_AT_EXEC (length) macro	The data for the parameter is sent with **SQLPutData**. If the *fSqlType* argument is SQL_LONGVARBINARY, SQL_LONGVARCHAR, or some other long Informix SQL data type, and the driver returns "Y" for the SQL_NEED_LONG_DATA_LEN information type in **SQLGetInfo**, length is the number of bytes of data to be sent for the parameter. Otherwise, length must be a nonnegative value and is ignored. For more information, see "Passing Parameter Values" on page 12-33. For example, to send 10,000 bytes of data with **SQLPutData** for an SQL_LONGVARCHAR parameter, an application sets *pcbValue* to SQL_LEN_DATA_AT_EXEC(10000). This macro was introduced in ODBC 2.0.

If *pcbValue* is a null pointer, the driver assumes that all input parameter values are non-null and that character and binary data are null-terminated. If *fParamType* is SQL_PARAM_OUTPUT and *rgbValue* and *pcbValue* are both null pointers, the driver discards the output value.

Important: *Application developers are strongly discouraged from specifying a null pointer for pcbValue when the parameter's INFORMIX-CLI C data type is SQL_C_BINARY. For SQL_C_BINARY data, the driver sends only the data preceding an occurrence of the null-termination character, 0x00. To ensure that the driver does not unexpectedly truncate SQL_C_BINARY data, pcbValue should contain a pointer to a valid length value.*

If the *fParamType* argument is SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT, *pcbValue* points to a buffer in which the driver returns SQL_NULL_DATA, the number of bytes available to return in *rgbValue* (excluding the null-termination byte of character data), or SQL_NO_TOTAL if the number of bytes available to return cannot be determined. If the procedure returns one or more result sets, the *pcbValue* buffer is not guaranteed to be set until all the results are fetched.

Passing Parameter Values

An application can pass the value for a parameter either in the *rgbValue* buffer or with one or more calls to **SQLPutData**. Parameters whose data is passed with **SQLPutData** are known as *data-at-execution* parameters. These are commonly used to send data for SQL_LONGVARBINARY and SQL_LONGVARCHAR parameters and can be mixed with other parameters.

To pass parameter values

1. The application calls **SQLBindParameter** for each parameter to bind buffers for the value (*rgbValue* argument) and length (*pcbValue* argument) of the parameter. For data-at-execution parameters, *rgbValue* is an application-defined 32-bit value such as a parameter number or a pointer to data. The value is returned later and can be used to identify the parameter.
2. The application places values for input and input/output parameters in the *rgbValue* and *pcbValue* buffers.
   - For normal parameters, the application places the parameter value in the *rgbValue* buffer and the length of that value in the *pcbValue* buffer.
   - For data-at-execution parameters, the application places the result of the SQL_LEN_DATA_AT_EXEC(*length*) macro in the *pcbValue* buffer.
3. The application calls **SQLExecute** or **SQLExecDirect** to execute the SQL statement.
   - If no data-at-execution parameters exist, the process is complete.
   - If any data-at-execution parameters exist, the function returns SQL_NEED_DATA.
4. The application calls **SQLParamData** to retrieve the application-defined value specified in the *rgbValue* argument for the first data-at-execution parameter to be processed.

   The data-at-execution parameters are similar to data-at-execution columns, although the value that **SQLParamData** returns is different for each.
5. The application calls **SQLPutData** one or more times to send data for the parameter. More than one call is needed if the data value is larger than the *rgbValue* buffer specified in **SQLPutData**; multiple calls to **SQLPutData** for the same parameter are allowed only when sending character C data or binary C data.

6. The application calls **SQLParamData** again to signal that all data was sent for the parameter.
   - If more data-at-execution parameters exist, **SQLParamData** returns SQL_NEED_DATA and the application-defined value for the next data-at-execution parameter to be processed. The application repeats steps 5 and 6.
   - If no more data-at-execution parameters exist, the process is complete. If the statement executes successfully, **SQLParamData** returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO; if the execution fails, it returns SQL_ERROR. At this point, **SQLParamData** can return any SQLSTATE that can be returned by the function used to execute the statement (**SQLExecDirect** or **SQLExecute**).

     Output values for any input/output or output parameters are available in the *rgbValue* and *pcbValue* buffers after the application retrieves any result sets that the statement generates.

If after **SQLExecute** or **SQLExecDirect** returns SQL_NEED_DATA and before data is sent for all data-at-execution parameters, the statement is canceled or an error occurs in **SQLParamData** or **SQLPutData**, the application can call only **SQLCancel**, **SQLGetFunctions**, **SQLParamData**, or **SQLPutData** with the *hstmt* or the *hdbc* associated with the *hstmt*. If the application calls any other function with the *hstmt* or the *hdbc* associated with the *hstmt*, the function returns SQL_ERROR and SQLSTATE S1010 (Function-sequence error).

If the application calls **SQLCancel** while the driver still needs data for data-at-execution parameters, the driver stops executing the statement; the application can then call **SQLExecute** or **SQLExecDirect** again. If the application calls **SQLParamData** or **SQLPutData** after it cancels the statement, the function returns SQL_ERROR and SQLSTATE S1008 (Operation canceled).

Code Example

In the following example, an application prepares an SQL statement to insert data into the **EMPLOYEE** table. The SQL statement contains parameters for the **NAME**, **AGE**, and **BIRTHDAY** columns. For each parameter in the statement, the application calls **SQLBindParameter** to specify the parameter's INFORMIX-CLI C data type and INFORMIX-CLI SQL data type, and to bind a buffer to each parameter. For each row of data, the application assigns data values to each parameter and calls **SQLExecute** to execute the statement.

```
#define NAME_LEN 30

UCHAR       szName[NAME_LEN];
SWORD       sAge;
SDWORD      cbName = SQL_NTS, cbAge = 0, cbBirthday = 0;
DATE_STRUCT dsBirthday;

retcode = SQLPrepare(hstmt,
    "INSERT INTO EMPLOYEE (NAME, AGE, BIRTHDAY) VALUES (?, ?, ?)",
    SQL_NTS);

if (retcode = = SQL_SUCCESS) {

    /* Specify data types and buffers.          */
    /* for Name, Age, Birthday parameter data.  */

    SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
                     SQL_CHAR, NAME_LEN, 0, szName, 0, &cbName);
    SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_SSHORT,
                     SQL_SMALLINT, 0, 0, &sAge, 0, &cbAge);
    SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_DATE,
                     SQL_DATE, 0, 0, &dsBirthday, 0, &cbBirthday);
    strcpy(szName, "Smith, John D.");    /* Specify first row of   */
    sAge = 40;                           /* parameter data         */
    dsBirthday.year = 1952;
    dsBirthday.month = 2;
    dsBirthday.day = 29;
    retcode = SQLExecute(hstmt);         /* Execute statement with */
                                         /* first row              */

    strcpy(szName, "Jones, Bob K.");     /* Specify second row of  */
    sAge = 52;                           /* parameter data         */
    dsBirthday.year = 1940;
    dsBirthday.month = 3;
    dsBirthday.day = 31;
    SQLExecute(hstmt);                   /* Execute statement with  */
                                         /* second row              */
}
```

For a similar example, see **SQLPutData**.

Related Functions

For Information About	See
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Returning the number of statement parameters	**SQLNumParams**
Returning the next parameter to send data for	**SQLParamData**
Sending parameter data at execution time	**SQLPutData**

♦

Level 2

SQLBrowseConnect

SQLBrowseConnect supports an iterative method of discovering and enumerating the attributes and attribute values required to connect to a data source. Each call to **SQLBrowseConnect** returns successive levels of attributes and attribute values. When all levels are enumerated, a connection to the data source is completed, and **SQLBrowseConnect** returns a complete connection string. A return code of SQL_SUCCESS_WITH_INFO or SQL_SUCCESS indicates that all connection information was specified, and the application is now connected to the data source.

Syntax

```
RETCODE SQLBrowseConnect(hdbc, szConnStrIn, cbConnStrIn,
szConnStrOut, cbConnStrOutMax, pcbConnStrOut)
```

The **SQLBrowseConnect** function accepts the following arguments.

Typedef	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.
UCHAR FAR *	*szConnStrIn*	Input	Browse request connection string. For more information, see "szConnStrIn Argument" on page 12-43.
SWORD	*cbConnStrIn*	Input	Length of *szConnStrIn*.

(1 of 2)

Typedef	Argument	Use	Description
UCHAR FAR *	*szConnStrOut*	Output	Pointer to storage for the browse result connection string. For more information, see "szConnStrOut Argument" on page 12-43.
SWORD	*cbConnStrOutMax*	Input	Maximum length of the *szConnStrOut* buffer.
SWORD FAR *	*pcbConnStrOut*	Output	The total number of bytes (excluding the null-termination byte) available to return in *szConnStrOut*. If the number of bytes available to return is greater than or equal to *cbConnStrOutMax*, the connection string in *szConnStrOut* is truncated to *cbConnStrOutMax* – 1 bytes.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The buffer *szConnStrOut* was not large enough to return entire browse result connection string, so the string was truncated. The argument *pcbConnStrOut* contains the length of the untruncated browse result connection string (function returns SQL_SUCCESS_WITH_INFO).
01S00	Invalid connection string attribute	An invalid attribute keyword was specified in the browse request connection string (*szConnStrIn*). (Function returns SQL_NEED_DATA.) An attribute keyword was specified in the browse request connection string (*szConnStrIn*) that does not apply to the current connection level (function returns SQL_NEED_DATA).
08001	Unable to connect to data source	The driver could not establish a connection with the data source.
08002	Connection in use	(DM) The specified *hdbc* already established a connection with a data source, and the connection is open.

(1 of 3)

SQLSTATE	Error	Description
08004	Data source rejected establishment of connection	The data source rejected the establishment of the connection for implementation-defined reasons.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
28000	Invalid authorization specification	Either the user identifier or the authorization string or both as specified in the browse-request connection string (*szConnStrIn*) violated restrictions defined by the data source.
IM002	Data source not found and no default driver specified	(DM) INFORMIX-CLI cannot find the data-source name specified in the browse-request connection string (*szConnStrIn*) in the **odbc.ini** file, and a default driver specification does not exist. (DM) INFORMIX-CLI cannot find the **odbc.ini** file.
IM003	Specified driver could not be loaded	(DM) The driver listed in the data source specification in the **odbc.ini** file, specified by the DRIVER keyword, was not found or could not be loaded for some other reason.
IM004	Driver **SQLAllocEnv** failed	(DM) During **SQLBrowseConnect**, the driver manager called the driver **SQLAllocEnv** function, and the driver returned an error.
IM005	Driver **SQLAllocConnect** failed	(DM) During **SQLBrowseConnect**, the driver manager called the driver **SQLAllocConnect** function, and the driver returned an error.
IM006	Driver **SQLSetConnectOption** failed	(DM) During **SQLBrowseConnect**, the driver manager called the driver **SQLSetConnectOption** function, and the driver returned an error.
IM009	Unable to load translation-shared library	The driver did not load the translation-shared library that was specified for the data source or for the connection.

(2 of 3)

SQLSTATE	Error	Description
IM010	Data-source name too long	(DM) The attribute value for the DSN keyword was longer than SQL_MAX_DSN_LENGTH characters.
IM011	Driver name too long	(DM) The attribute value for the DRIVER keyword was longer than 255 characters.
IM012	DRIVER keyword syntax error	(DM) The keyword-value pair for the DRIVER keyword contained a syntax error.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	(DM) The driver manager did not allocate memory required to support executing or completing the function. The driver did not allocate memory required to support executing or completing the function.
S1090	Invalid string or buffer length	(DM) The value specified for argument *cbConnStrIn* was less than 0 and was not equal to SQL_NTS. (DM) The value specified for argument *cbConnStrOutMax* was less than 0.
S1T00	Time-out expired	The time-out period expired before the connection to the data source completed. The time-out period is set through **SQLSetConnectOption**, SQL_LOGIN_TIMEOUT.

(3 of 3)

Usage

Each time an application calls **SQLBrowseConnect**, the function validates the current attributes, returns the next level of attributes, and returns a user-friendly name for each attribute. It might also return a list of valid values for those attributes. After an application has specified each level of attributes and values, **SQLBrowseConnect** connects to the data source and returns a complete connection string. This string can be used with **SQLDriverConnect** to reconnect to the data source.

szConnStrIn Argument

For information about INFORMIX-CLI connection strings, see "Connection Strings" on page 4-7.

szConnStrOut Argument

The browse-result connection string is a list of connection attributes. A connection attribute consists of an attribute keyword and a corresponding attribute value. The browse-result connection string has the following syntax:

```
connection-string ::= attribute[;] | attribute; connection-string
attribute ::= [*]attribute-keyword=attribute-value
attribute-keyword ::= ODBC-attribute-keyword
                                | driver-defined-attribute-keyword
ODBC-attribute-keyword = {UID | PWD}[:localized-identifier]
driver-defined-attribute-keyword ::= identifer[:localized-identifier]
attribute-value ::= {attribute-value-list} | ?
(The braces are literal; they are returned by the driver.)
attribute-value-list ::= character-string | character-string, attribute-value-
list
```

In this example, *character-string* has zero or more characters, *identifier* and *localized-identifier* have one or more characters, *attribute-keyword* is not case sensitive, and *attribute-value* might be case sensitive. Because of connection-string and initialization-file grammar, avoid keywords, localized identifiers, and attribute values that contain the characters [ ]{ }( ),;?*=!@\. Because of the registry grammar, keywords and data-source names cannot contain a backslash (\).

The browse-result connection string syntax is used according to the following semantic rules:

- If an asterisk (*) precedes an *attribute-keyword*, the *attribute* is optional and can be omitted in the next call to **SQLBrowseConnect**.
- An *attribute-keyword* names the kind of attribute for which an attribute value can be supplied. For information on attribute keywords that are available with INFORMIX-CLI, see "Connection Strings" on page 4-7.
- The {*attribute-value-list*} is an enumeration of actual values valid for the corresponding *attribute-keyword*. The braces ({ }) do not indicate a list of choices; the driver returns them. For example, the list might include server names or a list of database names.
- If the *attribute-value* is a single question mark (?), a single value corresponds to the *attribute-keyword*. For example, UID=JohnS; `PWD=Sesame`.
- Each call to **SQLBrowseConnect** returns only the information required to satisfy the next level of the connection process. The driver associates state information with the connection handle so that the context can be determined on each call.

Using SQLBrowseConnect

SQLBrowseConnect requires an allocated *hdbc*. The driver manager loads the driver that is specified in or that corresponds to the data-source name specified in the initial browse-request connection string. For additional information, see "Usage" on page 12-79. It might establish a connection with the data source during the browsing process. If **SQLBrowseConnect** returns SQL_ERROR, outstanding connections terminate, and the *hdbc* returns to an unconnected state.

When **SQLBrowseConnect** is called for the first time on an *hdbc*, the browse-request connection string must contain the DSN keyword or the DRIVER keyword. If the browse-request connection string contains the DSN keyword, the driver manager locates a corresponding data-source specification. If the driver manager cannot find the corresponding data-source specification, and no default data-source specification exists, it returns SQL_ERROR with SQLSTATE IM002 (Data source not found and no default driver specified).

If the browse-request connection string contains the DRIVER keyword, the driver manager loads the specified driver; it does not attempt to locate a data source. Because the DRIVER keyword does not use information from the **odbc.ini** file, the driver must define enough keywords so a driver can connect to a data source using only the information in the browse-request connection strings.

On each call to **SQLBrowseConnect**, the application specifies the connection attribute values in the browse-request connection string. The driver returns successive levels of attributes and attribute values in the browse-result connection string; it returns SQL_NEED_DATA as long as there are connection attributes that have not yet been enumerated in the browse-request connection string. The application uses the contents of the browse-result connection string to build the browse-request connection string for the next call to **SQLBrowseConnect**. The application cannot use the contents of previous browse-result connection strings when it builds the current one; that is, it cannot specify different values for attributes set in previous levels.

When all levels of connection and their associated attributes are enumerated, the driver returns SQL_SUCCESS, the connection to the data source is complete, and a complete connection string returns to the application.

The connection string can be used with **SQLDriverConnect** with the SQL_DRIVER_NOPROMPT option to establish another connection.

SQLBrowseConnect also returns SQL_NEED_DATA if recoverable, nonfatal errors occur during the browse process. For example, an invalid password supplied by the application or an invalid attribute keyword supplied by the application cause recoverable, nonfatal errors. When SQL_NEED_DATA returns and the browse-result connection string is unchanged, an error has occurred, and the application must call **SQLError** to return the SQLSTATE for browse-time errors. This permits the application to correct the attribute and continue the browse.

An application can terminate the browse process at any time by calling **SQLDisconnect**. The driver terminates any outstanding connections and returns the *hdbc* to an unconnected state.

For more information, see "Connection Strings" on page 4-7.

Code Example

In the following example, an application calls **SQLBrowseConnect** repeatedly. Each time **SQLBrowseConnect** returns SQL_NEED_DATA, it passes back information about the data that it needs in *szConnStrOut*. The application passes *szConnStrOut* to its routine **GetUserInput** (not shown). **GetUserInput** parses the information, builds and displays a dialog box, and returns the information entered by the user in *szConnStrIn*. The application passes the user's information to the driver in the next call to **SQLBrowseConnect**. After the application provides all the necessary information for the driver to connect to the data source, **SQLBrowseConnect** returns SQL_SUCCESS, and the application proceeds.

For example, to connect to the data source **My Source**, the following actions might occur. First, the application passes the following string to **SQLBrowseConnect**, as the following example shows:

```
"DSN=My Source"
```

The driver manager loads the driver associated with the data source **My Source**. It then calls the driver **SQLBrowseConnect** function with the same arguments that it received from the application. The driver returns the following string in *szConnStrOut*:

```
"HOST:Server={red,blue,green};UID:ID=?;PWD:Password=?"
```

The application passes this string to its **GetUserInput** routine, which provides a dialog box that asks the user to select the red, blue, or green database server and to enter a user ID and password. The routine passes the following user-specified information back in *szConnStrIn,* which the application passes to **SQLBrowseConnect**:

```
"HOST=red;UID=Smith;PWD=Sesame"
```

SQLBrowseConnect uses this information to connect to the red database server as **Smith** with the password **Sesame** and returns the following string in *szConnStrOut*:

```
"*DATABASE:Database={master,model,empdata}"
```

The application passes this string to its **GetUserInput** routine, which provides a dialog box that asks the user to select a database. The user selects **empdata**, and the application calls **SQLBrowseConnect** a final time with the following string:

```
"DATABASE=empdata"
```

This is the final piece of information that the driver needs to connect to the data source; **SQLBrowseConnect** returns SQL_SUCCESS, and *szConnStrOut* contains the completed connection string.

```
"DSN=My Source;HOST=red;UID=Smith;PWD=Sesame;DATABASE=empdata"

#define BRWS_LEN 100
HENV    henv;
HDBC    hdbc;
HSTMT   hstmt;
RETCODE retcode;
UCHAR   szConnStrIn[BRWS_LEN], szConnStrOut[BRWS_LEN];
SWORD   cbConnStrOut;

retcode = SQLAllocEnv(&henv);                 /* Environment handle */
if (retcode = = SQL_SUCCESS) {
    retcode = SQLAllocConnect(henv, &hdbc);    /* Connection handle  */
    if (retcode = = SQL_SUCCESS) {/* Call SQLBrowseConnect until it returns a
value other than */
        /* SQL_NEED_DATA (pass the data source name the first time). */
        /* If SQL_NEED_DATA is returned, call GetUserInput (not      */
        /* shown) to build a dialog from the values in szConnStrOut. */
        /* The user-supplied values are returned in szConnStrIn,     */
        /* which is passed in the next call to SQLBrowseConnect.     */

        lstrcpy(szConnStrIn, "DSN=MyServer");
        do {
            retcode = SQLBrowseConnect(hdbc, szConnStrIn, SQL_NTS,
                                    szConnStrOut, BRWS_LEN, &cbConnStrOut)
            if (retcode = = SQL_NEED_DATA)
                GetUserInput(szConnStrOut, szConnStrIn);
        } while (retcode = = SQL_NEED_DATA);

        if (retcode = = SQL_SUCCESS || retcode = = SQL_SUCCESS_WITH_INFO){

            /* Process data after successful connection */
            retcode = SQLAllocStmt(hdbc, &hstmt);
            if (retcode = = SQL_SUCCESS) {
                ...;
                ...;
                ...;
                SQLFreeStmt(hstmt, SQL_DROP);
            }
            SQLDisconnect(hdbc);
        }
    }
    SQLFreeConnect(hdbc);
}
SQLFreeEnv(henv);
```

Related Functions

For Information About	See
Allocating a connection handle	**SQLAllocConnect**
Connecting to a data source	**SQLConnect**
Disconnecting from a data source	**SQLDisconnect**
Connecting to a data source using a connection string or dialog box	**SQLDriverConnect**
Returning driver descriptions and attributes	**SQLDrivers**
Freeing a connection handle	**SQLFreeConnect**

♦

Core

SQLCancel

SQLCancel cancels the processing on an *hstmt* or a query.

Syntax

```
RETCODE SQLCancel(hstmt)
```

The **SQLCancel** function accepts the following arguments.

Typedef	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
70100	Operation aborted	The data source did not process the cancel request.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.

Usage

SQLCancel can cancel function processing on an *hstmt* that needs data.

If an application calls **SQLCancel** when processing does not involve the *hstmt*, **SQLCancel** has the same effect as **SQLFreeStmt** with the SQL_CLOSE option; this behavior is defined only for completeness, and applications should call **SQLFreeStmt** to close cursors.

Canceling Functions that Need Data

After **SQLExecute** or **SQLExecDirect** returns SQL_NEED_DATA and before data is sent for all data-at-execution parameters, an application can call **SQLCancel** to cancel the statement execution. After the statement is canceled, the application can call **SQLExecute** or **SQLExecDirect** again. For more information, see "Passing Parameter Values" on page 12-33.

Canceling Functions in Multithreaded Applications

In a multithreaded application, the application can cancel a function that runs synchronously on an *hstmt*. To cancel the function, the application calls **SQLCancel** with the same *hstmt* that the target function uses, but on a different thread. The return code of **SQLCancel** indicates whether or not the driver processed the request successfully. The return code of the original function indicates whether the function completed normally or was canceled.

Related Functions

For Information About	See
Assigning storage for a parameter	**SQLBindParameter**
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Freeing a statement handle	**SQLFreeStmt**
Returning the next parameter for which to send data	**SQLParamData**
Sending parameter data at execution time	**SQLPutData**

♦

Core

SQLColAttributes

SQLColAttributes returns descriptor information for a column in a result set; it cannot be used to return information about the bookmark column (column 0). Descriptor information is returned as a character string, a 32-bit descriptor-dependent value, or an integer value.

Syntax

```
RETCODE SQLColAttributes(hstmt, icol, fDescType, rgbDesc,
cbDescMax, pcbDesc, pfDesc)
```

The **SQLColAttributes** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UWORD	*icol*	Input	Column number of result data, ordered sequentially from left to right, starting at 1. Columns can be described in any order.
UWORD	*fDescType*	Input	A valid descriptor type. (See "Usage" on page 12-56.)
PTR	*rgbDesc*	Output	Pointer to storage for the descriptor information. The format of the descriptor information returned depends on the *fDescType*.
SWORD	*cbDescMax*	Input	Maximum length of the *rgbDesc* buffer.

(1 of 2)

Type	Argument	Use	Description
SWORD FAR *	*pcbDesc*	Output	Total number of bytes (excluding the null-termination byte for character data) available to return in *rgbDesc*. For character data, if the number of bytes available to return is greater than or equal to *cbDescMax*, the descriptor information in *rgbDesc* is truncated to *cbDescMax* – 1 bytes and is null-terminated by the driver. For all other types of data, the value of *cbValueMax* is ignored and the driver assumes the size of *rgbValue* is 32 bits.
SDWORD FAR *	*pfDesc*	Output	Pointer to an integer value to contain descriptor information for numeric descriptor types, such as SQL_COLUMN_LENGTH.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The buffer *rgbDesc* was not large enough to return the entire string value, so the string value was truncated. The argument *pcbDesc* contains the length of the untruncated string value (function returns SQL_SUCCESS_WITH_INFO).
24000	Invalid cursor state	The statement associated with the *hstmt* does not return a result set. There are no columns to describe.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support executing or completing the function.
S1002	Invalid column number	(DM) The value specified for the argument *icol* was 0, and the argument *fDescType* was not SQL_COLUMN_COUNT. The value specified for the argument *icol* was greater than the number of columns in the result set, and the argument *fDescType* was not SQL_COLUMN_COUNT.

(1 of 2)

SQLSTATE	Error	Description
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1010	Function-sequence error	(DM) The function was called prior to calling **SQLPrepare** or **SQLExecDirect** for the *hstmt*. (DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value specified for the argument *cbDescMax* was less than 0.
S1091	Descriptor type out of range	(DM) The value specified for the argument *fDescType* was in the block of numbers reserved for ODBC descriptor types but was not valid for the version of ODBC supported by the driver.
S1C00	Driver not capable	The driver or data source does not support the value specified for the argument *fDescType*.
S1T00	Time-out expired	The time-out period expired before the data source returned the requested information. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

SQLColAttributes can return any SQLSTATE that can be returned by **SQLPrepare** or **SQLExecute** when it is called after **SQLPrepare** and before **SQLExecute,** depending on when the data source evaluates the SQL statement associated with the *hstmt*.

Usage

SQLColAttributes returns information either in *pfDesc* or in *rgbDesc*. Integer information is returned in *pfDesc* as a 32-bit, signed value; all other formats of information are returned in *rgbDesc*. When information is returned in *pfDesc*, the driver ignores *rgbDesc*, *cbDescMax*, and *pcbDesc*. When information is returned in *rgbDesc*, the driver ignores *pfDesc*.

This function is an extensible alternative to **SQLDescribeCol**. **SQLDescribeCol** returns a fixed set of descriptor information based on ANSI-89 SQL. **SQLColAttributes** allows access to the more extensive set of descriptor information available in ANSI SQL-92 and Informix extensions.

fDescType: Descriptor Types

The following table shows the descriptor types and the arguments in which information is returned for them. If a descriptor type does not apply to a driver or data source, then, unless otherwise stated, the driver returns 0 in *pcbDesc* or an empty string in *rgbDesc*.

fDescType	Information Returned in	Description
SQL_COLUMN_AUTO_INCREMENT	*pfDesc*	TRUE if the column is auto increment. FALSE if the column is not auto increment or is not numeric. Auto increment is valid for numeric data type columns only. An application can insert values into an auto-increment column but cannot update values in the column.
SQL_COLUMN_CASE_SENSITIVE	*pfDesc*	TRUE if the column is treated as case sensitive for collations and comparisons. FALSE if the column is not treated as case sensitive for collations and comparisons or is non-character.
SQL_COLUMN_COUNT	*pfDesc*	Number of columns available in the result set. The *icol* argument is ignored.

(1 of 4)

fDescType	Information Returned in	Description
SQL_COLUMN_DISPLAY_SIZE	*pfDesc*	Maximum number of characters required to display data from the column. For more information on display size, see "Precision, Scale, Length, and Display Size" on page B-5.
SQL_COLUMN_LABEL	*rgbDesc*	The column label or title. For example, a column named **EmpName** might be labeled Employee Name. If a column does not have a label, the column name is returned. If the column is unlabeled and unnamed, an empty string is returned.
SQL_COLUMN_LENGTH	*pfDesc*	The length in bytes of data transferred on an **SQLGetData** or **SQLFetch** operation if SQL_C_DEFAULT is specified. For numeric data, this size can be different than the size of the data stored on the data source. For more information, see "Precision, Scale, Length, and Display Size" on page B-5.
SQL_COLUMN_MONEY	*pfDesc*	TRUE if the column is MONEY data type. FALSE if the column is not MONEY data type.
SQL_COLUMN_NAME	*rgbDesc*	The column name. If the column is unnamed, an empty string is returned.
SQL_COLUMN_NULLABLE	*pfDesc*	SQL_NO_NULLS if the column does not accept null values. SQL_NULLABLE if the column accepts null values. SQL_NULLABLE_UNKNOWN if it is not known whether the column accepts null values.
SQL_COLUMN_OWNER_NAME	*rgbDesc*	The owner of the table that contains the column. The returned value is implementation defined if the column is an expression or if the column is part of a view. If the data source does not support owners or the owner name cannot be determined, an empty string is returned.
SQL_COLUMN_PRECISION	*pfDesc*	The precision of the column on the data source. For more information on precision, see "Precision, Scale, Length, and Display Size" on page B-5.

(2 of 4)

fDescType	Information Returned in	Description
SQL_COLUMN_QUALIFIER_NAME	*rgbDesc*	The qualifier of the table that contains the column. The returned value is implementation defined if the column is an expression or if the column is part of a view. If the data source does not support qualifiers or the qualifier name cannot be determined, an empty string is returned.
SQL_COLUMN_SCALE	*pfDesc*	The scale of the column on the data source. For more information on scale, see "Precision, Scale, Length, and Display Size" on page B-5.
SQL_COLUMN_SEARCHABLE	*pfDesc*	SQL_UNSEARCHABLE if the column cannot be used in a WHERE clause. SQL_LIKE_ONLY if the column can be used in a WHERE clause only with the LIKE predicate. SQL_ALL_EXCEPT_LIKE if the column can be used in a WHERE clause with all comparison operators except LIKE. SQL_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator. Columns of type SQL_LONGVARCHAR and SQL_LONGVARBINARY usually return SQL_LIKE_ONLY.
SQL_COLUMN_TABLE_NAME	*rgbDesc*	The name of the table that contains the column. The returned value is implementation defined if the column is an expression or if the column is part of a view. If the table name cannot be determined, an empty string is returned.
SQL_COLUMN_TYPE	*pfDesc*	The column's Informix SQL data type. The possible values are any of the Informix SQL data types listed in Appendix B, "Data Types." For more information about data types and conversions, see Appendix B.
SQL_COLUMN_TYPE_NAME	*rgbDesc*	Data-source-dependent data type name; for example, CHAR, VARCHAR, MONEY, LONG VARBINARY, or CHAR ( ) FOR BIT DATA. If the type is unknown, an empty string is returned.

(3 of 4)

fDescType	Information Returned in	Description
SQL_COLUMN_UNSIGNED	*pfDesc*	TRUE if the column is unsigned or not numeric. FALSE if the column is signed.
SQL_COLUMN_UPDATABLE	*pfDesc*	Column is described by the values for the defined constants: SQL_ATTR_READONLY SQL_ATTR_WRITE SQL_ATTR_READWRITE_UNKNOWN SQL_COLUMN_UPDATABLE describes the updatability of the column in the result set. Whether a column is updatable can be based on the data type, user privileges, and the definition of the result set itself. If it is unclear whether a column is updatable, SQL_ATTR_READWRITE_UNKNOWN should be returned.

(4 of 4)

IUS

Additional Descriptor Types for Universal Server

The following table shows the additional descriptor types for Universal Server and the arguments in which information is returned for them. If a descriptor type does not apply to a driver or data source, then, unless otherwise stated, the driver returns 0 in *pcbDesc* or an empty string in *rgbDesc*.

fDescType	Information Returned in	Description
SQL_INFX_ATTR_FLAGS	*pfDesc*	Type attribute flags. These flags indicate whether the column type is NULLABLE and/or is a DISTINCT type. Two macros, ISNULLABLE() and ISDISTINCT(), are provided to test the flag value.
SQL_INFX_ATTR_EXTENDED_TYPE_ ALIGNMENT	*pfDesc*	Extended type alignment. If the column is of a built-in data type, the function returns 0.
SQL_INFX_ATTR_EXTENDED_TYPE_CODE	*pfDesc*	Extended type ID. The database server assigns a unique extended ID to each UDT. If the column is of a built-in data type, the function returns 0.
SQL_INFX_ATTR_EXTENDED_TYPE_NAME	*rgbDesc*	Extended type name. For example, if a column is of type *circle* where *circle* is a UDT, then *circle* is returned. If the column is of a built-in data type, the function returns an empty string.
SQL_INFX_ATTR_EXTENDED_TYPE_OWNER	*rgbDesc*	Extended type owner name. For example, if the column is a UDT of type *circle* and the owner of the *circle* type is *cliuser*, then the function returns *cliuser*. If the column is of a built-in data type, the function returns an empty string.
SQL_INFX_ATTR_SOURCE_TYPE_CODE	*pfDesc*	Source type of DISTINCT type columns. If the column is of a built-in data type or an OPAQUE data type, the function returns 0.

♦

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning information about a column in a result set	**SQLDescribeCol**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**

♦

Level 2

SQLColumnPrivileges

SQLColumnPrivileges returns a list of columns and associated privileges for the specified table. The driver returns the information as a result set on the specified *hstmt*.

Syntax

```
RETCODE SQLColumnPrivileges(hstmt, szTableQualifier,
cbTableQualifier, szTableOwner, cbTableOwner, szTableName,
cbTableName, szColumnName, cbColumnName)
```

The **SQLColumnPrivileges** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szTableQualifier*	Input	Table qualifier. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different data sources, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbTableQualifier*	Input	Length of *szTableQualifier.*
UCHAR FAR *	*szTableOwner*	Input	Owner name. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different data sources, an empty string ("") denotes those tables that do not have owners.
SWORD	*cbTableOwner*	Input	Length of *szTableOwner.*
UCHAR FAR *	*szTableName*	Input	Table name.
SWORD	*cbTableName*	Input	Length of *szTableName.*
UCHAR FAR *	*szColumnName*	Input	String search pattern for column names.
SWORD	*cbColumnName*	Input	Length of *szColumnName.*

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver was unable to allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

(1 of 2)

SQLSTATE	Error	Description
S1090	Invalid string or buffer length	(DM) The value of one of the name-length arguments was less than 0 but not equal to SQL_NTS. The value of one of the name-length arguments exceeded the maximum-length value for the corresponding qualifier or name. See "Usage" on page 12-56.
S1C00	Driver not capable	A table qualifier was specified, but the driver or data source does not support qualifiers. A table owner was specified, but the driver or data source does not support owners. A string search pattern was specified for the column name, but the data source does not support search patterns for that argument. The combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options was not supported by the driver or data source.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

SQLColumnPrivileges returns the results as a standard result set, ordered by TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, COLUMN_NAME, GRANTOR, and GRANTEE. The following table lists the columns in the result set.

Important: ***SQLColumnPrivileges*** *might not return all columns. For example, the driver might not return information about pseudocolumns, such as Informix ROWID. Applications can use any valid column, regardless of whether it is returned by* ***SQLColumnPrivileges****.*

The lengths of VARCHAR columns shown in the table are maximums; the actual lengths depend on the data source. To determine the actual lengths of the TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, and COLUMN_NAME columns, an application can call **SQLGetInfo** with the SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

Column Name	Data Type	Comments
TABLE_QUALIFIER	VARCHAR(128)	Table qualifier identifier; NULL if not applicable to the data source. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different data sources, it returns an empty string ("") for those tables that do not have qualifiers.
TABLE_OWNER	VARCHAR(128)	Table owner identifier; NULL if not applicable to the data source. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different data sources, it returns an empty string ("") for those tables that do not have owners.
TABLE_NAME	VARCHAR(128) not NULL	Table identifier.
COLUMN_NAME	VARCHAR(128) not NULL	Column identifier.
GRANTOR	VARCHAR(128)	Identifier of the user who granted the privilege; NULL if not applicable to the data source.

(1 of 2)

Column Name	Data Type	Comments
GRANTEE	VARCHAR(128) not NULL	Identifier of the user to whom the privilege was granted.
PRIVILEGE	VARCHAR(128) not NULL	Identifies the column privilege. Can be one of the following or others supported by the data source when implementation defined: ■ SELECT: The grantee is permitted to retrieve data for the column. ■ INSERT: The grantee is permitted to provide data for the column in new rows that are inserted into the associated table. ■ UPDATE: The grantee is permitted to update data in the column. ■ REFERENCES: The grantee is permitted to refer to the column within a constraint (for example, a unique, referential, or table-check constraint).
IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users; “YES”, “NO”, or NULL if unknown or not applicable to the data source.

(2 of 2)

The *szColumnName* argument accepts a search pattern. For more information about valid search patterns, see “Search Pattern Arguments” on page 12-8.

Code Example

For a code example of a similar function, see **SQLColumns**.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning the columns in a table or tables	**SQLColumns**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning privileges for a table or tables	**SQLTablePrivileges**
Returning a list of tables in a data source	**SQLTables**

♦

Level 1

SQLColumns

SQLColumns returns the list of column names in specified tables. The driver returns this information as a result set on the specified *hstmt*.

Syntax

```
RETCODE SQLColumns(hstmt, szTableQualifier,
cbTableQualifier, szTableOwner, cbTableOwner, szTableName,
cbTableName, szColumnName, cbColumnName)
```

The **SQLColumns** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szTableQualifier*	Input	Qualifier name. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different data sources, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbTableQualifier*	Input	Length of *szTableQualifier.*
UCHAR FAR *	*szTableOwner*	Input	String search pattern for owner names. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different data sources, an empty string ("") denotes those tables that do not have owners.
SWORD	*cbTableOwner*	Input	Length of *szTableOwner.*
UCHAR FAR *	*szTableName*	Input	String search pattern for table names.
SWORD	*cbTableName*	Input	Length of *szTableName.*
UCHAR FAR *	*szColumnName*	Input	String search pattern for column names.
SWORD	*cbColumnName*	Input	Length of *szColumnName.*

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support executing or completing the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.

(1 of 2)

SQLSTATE	Error	Description
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value of one of the name length arguments was less than 0 but not equal to SQL_NTS. The value of one of the name-length arguments exceeded the maximum-length value for the corresponding qualifier or name. The maximum length of each qualifier or name can be obtained by calling **SQLGetInfo** with the *fInfoType* values. For more information, see "Usage" on page 12-187.
S1C00	Driver not capable	A table qualifier was specified, but the driver or data source does not support qualifiers. A table owner was specified, but the driver or data source does not support owners. A string search pattern was specified for the table owner, table name, or column name, but the data source does not support search patterns for one or more of those arguments. The combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options is not supported by the driver or data source.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

This function is typically used before statement execution to retrieve information about columns for a table or tables from the catalog of the data source. In contrast, **SQLColAttributes** and **SQLDescribeCol** describe the columns in a result set, and **SQLNumResultCols** returns the number of columns in a result set.

SQLColumns returns the results as a standard result set, ordered by TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, COLUMN_NAME, DATA_TYPE, and S TYPE_NAME. The following table lists the columns in the result set. Additional columns beyond column 12 (REMARKS) can be defined by the driver.

The lengths of VARCHAR columns shown in the following table are maximums; the actual lengths depend on the data source. To determine the actual lengths of the TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, and COLUMN_NAME columns, an application can call **SQLGetInfo** with the SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

Column Name	Data Type	Comments
TABLE_QUALIFIER	VARCHAR(128)	Table qualifier identifier; null if not applicable to the data source. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different data sources, it returns an empty string ("") for those tables that do not have qualifiers.
TABLE_OWNER	VARCHAR(128)	Table owner identifier; null if not applicable to the data source. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different data sources, it returns an empty string ("") for those tables that do not have owners.
TABLE_NAME	VARCHAR(128) not NULL	Table identifier.

(1 of 3)

Column Name	Data Type	Comments
COLUMN_NAME	VARCHAR(128) not NULL	Column identifier.
DATA_TYPE	SMALLINT not NULL	Informix SQL data type. The possible values are any of the Informix SQL data types listed in Appendix B, "Data Types." For more information about data types and conversions, see Appendix B.
TYPE_NAME	VARCHAR(128) not NULL	Data-source-dependent data-type name; for example, CHAR, VARCHAR, MONEY, LONG VARBINARY, or CHAR ( ) for BIT DATA.
PRECISION	INTEGER	The precision of the column on the data source. For precision information, see "Precision, Scale, Length, and Display Size" on page B-5.
LENGTH	INTEGER	The length in bytes of data transferred on an **SQLGetData** or **SQLFetch** operation if SQL_C_DEFAULT is specified. For numeric data, this size might be different than the size of the data stored on the data source. This value is the same as the PRECISION column for character or binary data. For more information about length, see "Precision, Scale, Length, and Display Size" on page B-5.
SCALE		The scale of the column on the data source. For more scale information, see "Precision, Scale, Length, and Display Size" on page B-5. NULL is returned for data types where scale is not applicable.

(2 of 3)

Column Name	Data Type	Comments
RADIX	SMALLINT	For numeric data types, either 10 or 2. If it is 10, the values in PRECISION and SCALE give the number of decimal digits allowed for the column. For example, a DECIMAL(12,5) column would return a RADIX of 10, a PRECISION of 12, and a SCALE of 5; a FLOAT column could return a RADIX of 10, a PRECISION of 15, and a SCALE of NULL. If it is 2, the values in PRECISION and SCALE give the number of bits allowed in the column. For example, a FLOAT column could return a RADIX of 2, a PRECISION of 53, and a SCALE of NULL. NULL is returned for data types where RADIX is not applicable.
NULLABLE	SMALLINT not NULL	SQL_NO_NULLS if the column does not accept null values. SQL_NULLABLE if the column accepts null values. SQL_NULLABLE_UNKNOWN if it is not known if the column accepts null values.
REMARKS	VARCHAR(254)	A description of the column.

(3 of 3)

The *szTableOwner*, *szTableName*, and *szColumnName* arguments accept search patterns. For more information about valid search patterns, see “Search Pattern Arguments” on page 12-8.

Code Example

In the following example, an application calls **SQLColumns** to return a result set that describes each column in the **EMPLOYEE** table. It then calls **SQLBindCol** to bind the columns in the result set to the storage locations. Finally, the application fetches each row of data with **SQLFetch** and processes it.

```
#define STR_LEN 128+1
#define REM_LEN 254+1

/* Declare storage locations for result set data */

UCHAR  szQualifier[STR_LEN], szOwner[STR_LEN];
UCHAR  szTableName[STR_LEN], szColName[STR_LEN];
UCHAR  szTypeName[STR_LEN], szRemarks[REM_LEN];
SDWORD Precision, Length;
SWORD  DataType, Scale, Radix, Nullable;

/* Declare storage locations for bytes available to return */

SDWORD cbQualifier, cbOwner, cbTableName, cbColName;
SDWORD cbTypeName, cbRemarks, cbDataType, cbPrecision;
SDWORD cbLength, cbScale, cbRadix, cbNullable;

retcode = SQLColumns(hstmt,
                     NULL, 0,              /* All qualifiers */
                     NULL, 0,              /* All owners     */
                     "EMPLOYEE", SQL_NTS,  /* EMPLOYEE table */
                     NULL, 0);             /* All columns    */
if (retcode == SQL_SUCCESS) {

    /* Bind columns in result set to storage locations */

    SQLBindCol(hstmt, 1, SQL_C_CHAR, szQualifier, STR_LEN,&cbQualifier);
    SQLBindCol(hstmt, 2, SQL_C_CHAR, szOwner, STR_LEN, &cbOwner);
    SQLBindCol(hstmt, 3, SQL_C_CHAR, szTableName, STR_LEN,&cbTableName);
    SQLBindCol(hstmt, 4, SQL_C_CHAR, szColName, STR_LEN, &cbColName);
    SQLBindCol(hstmt, 5, SQL_C_SSHORT, &DataType, 0, &cbDataType);
    SQLBindCol(hstmt, 6, SQL_C_CHAR, szTypeName, STR_LEN, &cbTypeName);
    SQLBindCol(hstmt, 7, SQL_C_SLONG, &Precision, 0, &cbPrecision);
    SQLBindCol(hstmt, 8, SQL_C_SLONG, &Length, 0, &cbLength);
    SQLBindCol(hstmt, 9, SQL_C_SSHORT, &Scale, 0, &cbScale);
    SQLBindCol(hstmt, 10, SQL_C_SSHORT, &Radix, 0, &cbRadix);
    SQLBindCol(hstmt, 11, SQL_C_SSHORT, &Nullable, 0, &cbNullable);
```

```
        SQLBindCol(hstmt, 12, SQL_C_CHAR, szRemarks, REM_LEN, &cbRemarks);

            while(TRUE) {
            retcode = SQLFetch(hstmt);
            if (retcode == SQL_ERROR || retcode == SQL_SUCCESS_WITH_INFO) {
                show_error( );
            }
            if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){
                ...;  /* Process fetched data */
            } else {
                break;
            }
        }
    }
```

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning privileges for a column or columns	**SQLColumnPrivileges**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning table statistics and indexes	**SQLStatistics**
Returning a list of tables in a data source	**SQLTables**
Returning privileges for a table or tables	**SQLTablePrivileges**

♦

Core

SQLConnect

SQLConnect loads a driver and establishes a connection to a data source. The connection handle references where all information about the connection, including status, transaction state, and error information is stored.

Syntax

```
RETCODE SQLConnect(hdbc, szDSN, cbDSN, szUID, cbUID,
szAuthStr, cbAuthStr)
```

The **SQLConnect** function accepts the following arguments.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle
UCHAR FAR *	*szDSN*	Input	Data-source name
SWORD	*cbDSN*	Input	Length of *szDSN*
UCHAR FAR *	*szUID*	Input	User identifier
SWORD	*cbUID*	Input	Length of *szUID*
UCHAR FAR *	*szAuthStr*	Input	Authentication string (typically the password)
SWORD	*cbAuthStr*	Input	Length of *szAuthStr*

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08001	Unable to connect to data source	The driver could not establish a connection with the data source.
08002	Connection in use	(DM) The specified *hdbc* was already used to establish a connection with a data source, and the connection was still open.
08004	Data source rejected establishment of connection	The data source rejected the establishment of the connection for implementation-defined reasons.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
28000	Invalid authorization specification	The value specified for the argument *szUID* or the value specified for the argument *szAuthStr* violated restrictions defined by the data source.
IM002	Data source not found and no default driver specified	(DM) The data-source name specified in the argument *szDSN* was not found, nor was there a default driver specification. (DM) The **odbc.ini** file was not found.
IM003	Specified driver could not be loaded	(DM) The driver listed in the data source specification in the **odbc.ini** file was not found or could not be loaded for some other reason.
IM004	Driver **SQLAllocEnv** failed	(DM) During **SQLConnect**, the driver manager called the driver **SQLAllocEnv** function, and the driver returned an error.

(1 of 2)

SQLSTATE	Error	Description
IM005	Driver **SQLAllocConnect** failed	(DM) During **SQLConnect**, the driver manager called the driver **SQLAllocConnect** function, and the driver returned an error.
IM006	Driver **SQLSetConnectOption** failed	(DM) During **SQLConnect**, the driver manager called the driver **SQLSetConnectOption** function, and the driver returned an error (function returns SQL_SUCCESS_WITH_INFO).
IM009	Unable to load translation shared library	The driver did not load the translation shared library that was specified for the data source.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	(DM) The driver manager did not allocate memory required to support execution or completion of the function. The driver did not allocate memory required to support execution or completion of the function.
S1090	Invalid string or buffer length	(DM) The value specified for argument *cbDSN* was less than 0, but not equal to SQL_NTS. (DM) The value specified for argument *cbDSN* exceeded the maximum length for a data- source name. (DM) The value specified for argument *cbUID* was less than 0 but not equal to SQL_NTS. (DM) The value specified for argument *cbAuthStr* was less than 0 but not equal to SQL_NTS.
S1T00	Time-out expired	The time-out period expired before the connection to the data source completed. The time-out period is set through **SQLSetConnectOption**, SQL_LOGIN_TIMEOUT.

(2 of 2)

Usage

INFORMIX-CLI does not load its driver until the application calls a function (**SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**) to connect to the driver. Until that point, INFORMIX-CLI works with its own handles and manages connection information. When the application calls a connection function, INFORMIX-CLI checks to see if its driver is currently loaded:

- If the driver is not loaded, INFORMIX-CLI loads the driver and calls **SQLAllocEnv**, **SQLAllocConnect**, **SQLSetConnectOption** (if the application specified any connection options), and the connection function in the driver. INFORMIX-CLI returns SQLSTATE IM006 (Driver **SQLSetConnectOption** function failed) and SQL_SUCCESS_WITH_INFO for the connection function if the driver returns an error for **SQLSetConnectOption**.
- If the driver is already loaded on the *hdbc*, INFORMIX-CLI calls only the connection function in the driver. In this case, the driver must ensure that all connection options for the *hdbc* maintain their current settings.

The driver then allocates handles and initializes itself.

When the application calls **SQLDisconnect**, INFORMIX-CLI calls **SQLDisconnect** in the INFORMIX-CLI driver. However, INFORMIX-CLI does not unload the driver. By keeping the driver in memory, INFORMIX-CLI makes the driver available for applications that repeatedly connect to and disconnect from a data source. When the application calls **SQLFreeConnect**, INFORMIX-CLI calls **SQLFreeConnect** and **SQLFreeEnv** in the driver and then unloads the driver.

An INFORMIX-CLI application can establish more than one connection.

After being loaded, the INFORMIX-CLI driver can locate its corresponding data-source specification and use driver-specific information from the specification to complete its set of required connection information.

Code Example

In the following example, an application allocates environment and connection handles. It then connects to the **EmpData** data source with the user ID **JohnS** and the password **Sesame** and processes data. When it finishes processing data, it disconnects from the data source and frees the handles.

```
HENV    henv;
HDBC    hdbc;
HSTMT   hstmt;
RETCODE retcode;

retcode = SQLAllocEnv(&henv);              /* Environment handle */
if (retcode == SQL_SUCCESS) {
    retcode = SQLAllocConnect(henv, &hdbc); /* Connection handle */
    if (retcode == SQL_SUCCESS) {

        /* Connect to data source */

        retcode = SQLConnect(hdbc, "EmpData", SQL_NTS,

                             "JohnS", SQL_NTS,

                             "Sesame", SQL_NTS);

        if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){

            /* Process data after successful connection */

            retcode = SQLAllocStmt(hdbc, &hstmt); /* Statement handle */
            if (retcode == SQL_SUCCESS) {
                ...;
                ...;
                ...;
                SQLFreeStmt(hstmt, SQL_DROP);
            }
            SQLDisconnect(hdbc);
        }
        SQLFreeConnect(hdbc);
    }
    SQLFreeEnv(henv);
}
```

Related Functions

For Information About	See
Allocating a connection handle	**SQLAllocConnect**
Allocating a statement handle	**SQLAllocStmt**
Discovering and enumerating values required to connect to a data source	**SQLBrowseConnect**
Disconnecting from a data source	**SQLDisconnect**
Connecting to a data source using a connection string or dialog box	**SQLDriverConnect**
Returning the setting of a connection option	**SQLGetConnectOption**
Setting a connection option	**SQLSetConnectOption**

♦

Level 2

SQLDataSources

SQLDataSources lists data-source names.

Syntax

```
RETCODE SQLDataSources(henv, fDirection, szDSN, cbDSNMax,
pcbDSN, szDescription, cbDescriptionMax, pcbDescription)
```

The **SQLDataSources** function accepts the following arguments.

Type	Argument	Use	Description
HENV	*henv*	Input	Environment handle.
UWORD	*fDirection*	Input	Determines whether the function fetches the next data-source name in the list (SQL_FETCH_NEXT) or whether the search starts from the beginning of the list (SQL_FETCH_FIRST).
UCHAR FAR *	*szDSN*	Output	Pointer to storage for the data-source name.
SWORD	*cbDSNMax*	Input	Maximum length of the *szDSN* buffer; this buffer need not be longer than SQL_MAX_DSN_LENGTH + 1.
SWORD FAR *	*pcbDSN*	Output	Total number of bytes (excluding the null-termination byte) available to return in *szDSN*. If the number of bytes available to return is greater than or equal to *cbDSNMax*, the data-source name in *szDSN* is truncated to *cbDSNMax* – 1 bytes.

(1 of 2)

Type	Argument	Use	Description
UCHAR FAR *	*szDescription*	Output	Pointer to storage for the description of the driver associated with the data source.
SWORD	*cbDescriptionMax*	Input	Maximum length of the *szDescription* buffer; this buffer should be at least 255 bytes.
SWORD FAR *	*pcbDescription*	Output	Total number of bytes (excluding the null-termination byte) available to return in *szDescription*. If the number of bytes available to return is greater than or equal to *cbDescriptionMax*, the driver description in *szDescription* is truncated to *cbDescriptionMax* – 1 bytes.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	(DM) The buffer *szDSN* was not large enough to return the entire data-source name, so the name was truncated. The argument *pcbDSN* contains the length of the entire data-source name (function returns SQL_SUCCESS_WITH_INFO).
		(DM) The buffer *szDescription* was not large enough to return the entire driver description, so the description was truncated. The argument *pcbDescription* contains the length of the untruncated data-source description (function returns SQL_SUCCESS_WITH_INFO).
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	(DM) The driver manager did not allocate memory required to support execution or completion of the function.

(1 of 2)

SQLSTATE	Error	Description
S1090	Invalid string or buffer length	(DM) The value specified for argument *cbDSNMax* was less than 0.
		(DM) The value specified for argument *cbDescriptionMax* was less than 0.
S1103	Direction option out of range	(DM) The value specified for the argument *fDirection* was not equal to SQL_FETCH_FIRST or SQL_FETCH_NEXT.

(2 of 2)

Usage

An application can call **SQLDataSources** multiple times to retrieve all data-source names. When no more data-source names remain, the function returns SQL_NO_DATA_FOUND. If **SQLDataSources** is called with SQL_FETCH_NEXT immediately after it returns SQL_NO_DATA_FOUND, it returns the first data-source name.

If SQL_FETCH_NEXT is passed to **SQLDataSources** the first time that it is called, it returns the first data-source name.

INFORMIX-CLI determines how data-source names are mapped to actual data sources.

Related Functions

For Information About	See
Discovering and listing values required to connect to a data source	**SQLBrowseConnect**
Connecting to a data source	**SQLConnect**
Connecting to a data source using a connection string or dialog box	**SQLDriverConnect**
Returning driver descriptions and attributes	**SQLDrivers**

♦

Core

SQLDescribeCol

SQLDescribeCol returns the result descriptor (column name, type, precision, scale, and whether or not it can have a NULL value) for one column in the result set; it cannot be used to return information about the bookmark column (column 0).

Syntax

```
RETCODE SQLDescribeCol(hstmt, icol, szColName, cbColNameMax,
pcbColName, pfSqlType, pcbColDef, pibScale, pfNullable)
```

The **SQLDescribeCol** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UWORD	*icol*	Input	Column number of result data, ordered sequentially left to right, starting at 1.
UCHAR FAR *	*szColName*	Output	Pointer to storage for the column name. If the column is unnamed or the column name cannot be determined, the driver returns an empty string.
SWORD	*cbColNameMax*	Input	Maximum length of the *szColName* buffer.
SWORD FAR *	*pcbColName*	Output	Total number of bytes (excluding the null-termination byte) available to return in *szColName*. If the number of bytes available to return is greater than or equal to *cbColNameMax*, the column name in *szColName* is truncated to *cbColNameMax* – 1 bytes.

(1 of 2)

Type	Argument	Use	Description
SWORD FAR *	*pfSqlType*	Output	The column's INFORMIX-CLI SQL data type. The possible values are any of the *fSqlType* values listed in Appendix B, "Data Types." If the data type cannot be determined, the driver returns 0. For more information about data types and conversions, see Appendix B.
UDWORD FAR *	*pcbColDef*	Output	The precision of the column on the data source. If the precision cannot be determined, the driver returns 0. For more information on precision, see "Precision, Scale, Length, and Display Size" on page B-5.
SWORD FAR *	*pibScale*	Output	The scale of the column on the data source. If the scale cannot be determined or is not applicable, the driver returns 0. For more information on scale, see "Precision, Scale, Length, and Display Size" on page B-5.
SWORD FAR *	*pfNullable*	Output	Indicates whether the column allows one of the following values: ■ SQL_NO_NULLS: The column does not allow null values. ■ SQL_NULLABLE: The column allows null values. ■ SQL_NULLABLE_UNKNOWN: The driver cannot determine whether the column allows null values.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The buffer *szColName* was not large enough to return the entire column name, so the column name was truncated. The argument *pcbColName* contains the length of the untruncated column name (function returns SQL_SUCCESS_WITH_INFO).
24000	Invalid cursor state	The statement associated with the *hstmt* did not return a result set. There were no columns to describe.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support executing or completing the function.
S1002	Invalid column number	(DM) The value specified for the argument *icol* was 0. The value specified for the argument *icol* was greater than the number of columns in the result set.

(1 of 2)

SQLSTATE	Error	Description
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1010	Function-sequence error	(DM) The function was called prior to calling **SQLPrepare** or **SQLExecDirect** for the *hstmt*. (DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value specified for argument *cbColNameMax* was less than 0.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

SQLDescribeCol can return any SQLSTATE that **SQLPrepare** or **SQLExecute** returns when **SQLDescribeCol** is called after **SQLPrepare** and before **SQLExecute**, depending on when the data source evaluates the SQL statement associated with the *hstmt*.

Usage

An application typically calls **SQLDescribeCol** after a call to **SQLPrepare** and before or after the associated call to **SQLExecute**. An application can also call **SQLDescribeCol** after a call to **SQLExecDirect**.

SQLDescribeCol retrieves the column name, type, and length generated by a SELECT statement. If the column is an expression, *szColName* is either an empty string or a driver-defined name.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning information about a column in a result set	**SQLColAttributes**
Fetching a row of data	**SQLFetch**
Returning the number of result-set columns	**SQLNumResultCols**
Preparing a statement for execution	**SQLPrepare**

♦

Core

SQLDisconnect

SQLDisconnect closes the connection associated with a specific connection handle.

Syntax

```
RETCODE SQLDisconnect(hdbc)
```

The **SQLDisconnect** function accepts the following argument.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01002	Disconnect error	An error occurred during the disconnect. However, the disconnect succeeded (function returns SQL_SUCCESS_WITH_INFO).
08003	Connection not open	(DM) The connection specified in the argument *hdbc* was not open.
25000	Invalid transaction state	A transaction was in process on the connection specified by the argument *hdbc*. The transaction remains active.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* associated with the *hdbc* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

Usage

If an application calls **SQLDisconnect** after **SQLBrowseConnect** returns SQL_NEED_DATA and before it returns a different return code, the driver cancels the connection-browsing process and returns the *hdbc* to an unconnected state.

If an application calls **SQLDisconnect** while an incomplete transaction is associated with the connection handle, the driver returns SQLSTATE 25000 (Invalid transaction state), indicating that the transaction is unchanged and the connection is open. An incomplete transaction is one that was not committed or rolled back with **SQLTransact**.

If an application calls **SQLDisconnect** before it frees every *hstmt* associated with the connection, the driver frees each remaining *hstmt* after it successfully disconnects from the data source.

Code Example

See **SQLBrowseConnect** and **SQLConnect**.

Related Functions

For Information About	See
Allocating a connection handle	**SQLAllocConnect**
Connecting to a data source	**SQLConnect**
Connecting to a data source using a connection string or dialog box	**SQLDriverConnect**
Freeing a connection handle	**SQLFreeConnect**
Executing a commit or rollback operation	**SQLTransact**

♦

Level 1

SQLDriverConnect

SQLDriverConnect is an alternative to **SQLConnect**. It supports data sources that require more connection information than the three arguments in **SQLConnect** dialog boxes to prompt the user for all connection information, and data sources that are not defined data source names.

SQLDriverConnect provides the following connection options:

- You can establish a connection using a connection string that contains the data-source name, one or more user IDs, one or more passwords, and other information required by the data source.
- You can establish a connection using a partial connection string or no additional information; in this case, INFORMIX-CLI can prompt the user for connection information.

Once a connection is established, **SQLDriverConnect** returns the completed connection string. The application can use this string for subsequent connection requests.

Syntax

```
RETCODE SQLDriverConnect(hdbc, hwnd, szConnStrIn,
cbConnStrIn, szConnStrOut, cbConnStrOutMax, pcbConnStrOut,
fDriverCompletion)
```

The **SQLDriverConnect** function accepts the following arguments.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.
HWND	*hwnd*	Input	Window handle (platform dependent): For Windows, this is the parent window handle. For UNIX, this is the parent Widget handle. If the window handle is not applicable, or a null pointer is passed, **SQLDriverConnect** does not present any dialog boxes.
UCHAR FAR *	*szConnStrIn*	Input	A full connection string, a partial connection string, or an empty string.
SWORD	*cbConnStrIn*	Input	Length of *szConnStrIn*.

(1 of 2)

Type	Argument	Use	Description
UCHAR FAR	**szConnStrOut*	Output	Pointer to storage for the completed connection string. Upon successful connection to the target data source, this buffer contains the completed connection string. Applications should allocate at least 255 bytes for this buffer.
SWORD	*cbConnStrOutMax*	Input	Maximum length of the *szConnStrOut* buffer.
SWORD FAR *	*pcbConnStrOut*	Output	Pointer to the total number of bytes (excluding the null termination byte) available to return in *szConnStrOut*. If the number of bytes available to return is greater than or equal to *cbConnStrOutMax*, the completed connection string in *szConnStrOut* is truncated to *cbConnStrOutMax* – 1 bytes.
UWORD	*fDriverCompletion*	Input	Flag that indicates whether INFORMIX-CLI must prompt for more connection information: SQL_DRIVER_PROMPT, SQL_DRIVER_COMPLETE, SQL_DRIVER_COMPLETE_REQUIRED, or SQL_DRIVER_NOPROMPT. For more information, see "Driver Manager Guidelines" on page 12-99 and "Driver Guidelines" on page 12-100.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The buffer *szConnStrOut* was not large enough to return the entire connection string, so the connection string was truncated. The argument *pcbConnStrOut* contains the length of the untruncated connection string (function returns SQL_SUCCESS_WITH_INFO).
01S00	Invalid connection string attribute	An invalid attribute keyword was specified in the connection string (*szConnStrIn*), but the driver connected to the data source anyway (function returns SQL_SUCCESS_WITH_INFO).
08001	Unable to connect to data source	The driver did not establish a connection with the data source.
08002	Connection in use	(DM) The specified *hdbc* was used already to establish a connection with a data source, and the connection was still open.
08004	Data source rejected establishment of connection	The data source rejected the establishment of the connection for implementation-defined reasons.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.

(1 of 3)

SQLSTATE	Error	Description
28000	Invalid authorization specification	Either the user identifier or the authorization string or both as specified in the connection string (*szConnStrIn*) violated restrictions defined by the data source.
IM002	Data source not found and no default driver specified	(DM) The data-source name specified in the connection string (*szConnStrIn*) was not found, and no default driver specification existed.
IM003	Specified driver could not be loaded	(DM) The driver listed in the data- source specification, or specified by the DRIVER keyword, was not found or was not loaded for some other reason.
IM004	Driver **SQLAllocEnv** failed	(DM) During **SQLDriverConnect**, the driver manager called the driver **SQLAllocEnv** function, and the driver returned an error.
IM005	Driver **SQLAllocConnect** failed	(DM) During **SQLDriverConnect**, the driver manager called the driver **SQLAllocConnect** function, and the driver returned an error.
IM006	Driver **SQLSetConnectOption** failed	(DM) During **SQLDriverConnect**, the driver manager called the driver **SQLSetConnectOption** function, and the driver returned an error.
IM007	No data source or driver specified; dialog prohibited	C *fDriverCompletion* was SQL_DRIVER_NOPROMPT.
IM008	Dialog failed	(DM) The driver manager attempted to display the SQL Data Sources dialog box but failed. The driver attempted to display its login dialog box but failed.
IM009	Unable to load translation shared library	The driver did not load the translation shared library that was specified for the data source or for the connection.
IM010	Data-source name too long	(DM) The attribute value for the DSN keyword was longer than SQL_MAX_DSN_LENGTH characters.

(2 of 3)

SQLSTATE	Error	Description
IM011	Driver name too long	(DM) The attribute value for the DRIVER keyword was longer than 255 characters.
IM012	DRIVER keyword syntax error	(DM) The keyword-value pair for the DRIVER keyword contained a syntax error.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver manager did not allocate memory required to support execution or completion of the function. The driver did not allocate memory required to support execution or completion of the function.
S1090	Invalid string or buffer length	(DM) The value specified for argument *cbConnStrIn* was less than 0 and was not equal to SQL_NTS. (DM) The value specified for argument *cbConnStrOutMax* was less than 0.
S1110	Invalid driver completion	(DM) The value specified for the argument *fDriverCompletion* was not equal to SQL_DRIVER_PROMPT, SQL_DRIVER_COMPLETE, SQL_DRIVER_COMPLETE_REQUIRED, or SQL_DRIVER_NOPROMPT.
S1T00	Time-out expired	The time-out period expired before the connection to the data source completed. The time-out period is set through **SQLSetConnectOption**, SQL_LOGIN_TIMEOUT.

(3 of 3)

Usage

SQLDriverConnect uses a connection string to specify the information needed to connect to a driver and data source. The connection string is a more flexible interface than the data-source name, user ID, and password used by **SQLConnect**. The application can use the connection string for multiple levels of login authorization or to convey other data-source-specific connection information.

Driver Manager Guidelines

Tip: *On Windows, a driver manager is mandatory and INFORMIX-CLI provides the driver manager. On UNIX, a driver manager is optional. If you want to use a driver manager on UNIX, you need to purchase one from a third-party vendor.*

The driver manager constructs a connection string to pass to the driver in the *szConnStrIn* argument of the driver **SQLDriverConnect** function. The driver manager does not modify the *szConnStrIn* argument passed to it by the application.

If the connection string specified by the application contains the DSN keyword or does not contain either the DSN or DRIVER keywords, the action of the driver manager is based on the value of the *fDriverCompletion* argument, as the following list explains:

- SQL_DRIVER_PROMPT

  The driver manager displays the Data Sources dialog box. It constructs a connection string from the data source name returned by the dialog box and any other keywords passed to it by the application. If the data-source name returned by the dialog box is empty, the driver manager specifies the keyword-value pair DSN=Default.

- SQL_DRIVER_COMPLETE or SQL_DRIVER_COMPLETE_REQUIRED

  If the connection string specified by the application includes the DSN keyword, the driver manager copies the connection string specified by the application. Otherwise, it takes the same actions as it does when *fDriverCompletion* is SQL_DRIVER_PROMPT.

- SQL_DRIVER_NOPROMPT

  The driver manager copies the connection string specified by the application.

If the connection string specified by the application contains the DRIVER keyword, the driver manager copies the connection string specified by the application.

Using the connection string that it constructed, the driver manager determines which driver to use, loads that driver, and passes the connection string that it has constructed to the driver. For more information about the interaction of the driver manager and the driver, see "Usage" on page 12-79. If the connection string contains the DSN keyword or does not contain either the DSN or the DRIVER keyword, the driver manager determines which driver to use, as the following steps show:

1. If the connection string contains the DSN keyword, the driver manager retrieves the driver associated with the data source.
2. If the connection string does not contain the DSN keyword or the data source is not found, the driver manager retrieves the driver associated with the default data source. However, the driver manager does not change the value of the DSN keyword in the connection string.
3. If the data source is not found and the Default data source is not found, the driver manager returns SQL_ERROR with SQLSTATE IM002 (Data source not found and no default driver specified).

Driver Guidelines

The driver checks to see whether the connection string passed to it (by the driver manager or the application) contains the DSN or DRIVER keyword. If the connection string contains the DRIVER keyword, the driver cannot retrieve information about the data source. If the connection string contains the DSN keyword or does not contain either the DSN or the DRIVER keyword, the driver can retrieve information about the data source from the initialization files, as the following steps show:

1. If the connection string contains the DSN keyword, the driver retrieves the information for the specified data source.
2. If the connection string does not contain the DSN keyword or the specified data source is not found, the driver retrieves the information for the default data source.

The driver uses any information that it retrieves from the initialization files to augment the information passed to it in the connection string. If the information in the initialization files duplicates information in the connection string, the driver uses the information in the connection string.

Based on the value of *fDriverCompletion*, the driver prompts the user for connection information, such as the user ID and password, and connects to the data source, as follows:

- SQL_DRIVER_PROMPT

  The driver displays a dialog box, using the values from the connection string and the initialization files as initial values. When the user exits the dialog box, the driver connects to the data source. It also constructs a connection string from the value of the DSN or DRIVER keyword in *szConnStrIn* and the information returned from the dialog box. It places this connection string in the buffer referenced by *szConnStrOut*.

- SQL_DRIVER_COMPLETE or SQL_DRIVER_COMPLETE_REQUIRED

  If the connection string contains enough correct information, the driver connects to the data source and copies *szConnStrIn* to *szConnStrOut*. If any information is missing or incorrect, the driver takes the same actions as it does when *fDriverCompletion* is SQL_DRIVER_PROMPT, except that if *fDriverCompletion* is SQL_DRIVER_COMPLETE_REQUIRED, the driver disables the controls for any information that is not required to connect to the data source.

- SQL_DRIVER_NOPROMPT

  If the connection string contains enough information, the driver connects to the data source and copies *szConnStrIn* to *szConnStrOut*. Otherwise, the driver returns SQL_ERROR for **SQLDriverConnect**.

After successfully connecting to the data source, the driver also sets *pcbConnStrOut* to the length of *szConnStrOut*.

If the user cancels a dialog box presented by INFORMIX-CLI, **SQLDriverConnect** returns SQL_NO_DATA_FOUND.

For information about how the driver manager and the driver interact during the connection process, see "Usage" on page 12-79.

Connection Options

The driver opens the connection in SQL_MODE_READ_WRITE access mode by default. To set the access mode to SQL_MODE_READ_ONLY, the application must call **SQLSetConnectOption** with the SQL_ACCESS_MODE option before calling **SQLDriverConnect**.

Related Functions

For Information About	See
Allocating a connection handle	**SQLAllocConnect**
Discovering and enumerating values required to connect to a data source	**SQLBrowseConnect**
Connecting to a data source	**SQLConnect**
Disconnecting from a data source	**SQLDisconnect**
Returning driver descriptions and attributes	**SQLDrivers**
Freeing a connection handle	**SQLFreeConnect**
Setting a connection option	**SQLSetConnectOption**

♦

Level 2

SQLDrivers

SQLDrivers lists driver descriptions and driver-attribute keywords.

Syntax

```
RETCODE SQLDrivers(henv, fDirection, szDriverDesc,
cbDriverDescMax, pcbDriverDesc, szDriverAttributes,
cbDrvrAttrMax, pcbDrvrAttr)
```

The **SQLDrivers** function accepts the following arguments.

Type	Argument	Use	Description
HENV	*henv*	Input	Environment handle.
UWORD	*fDirection*	Input	Specifies whether the function fetches the next driver description in the list (SQL_FETCH_NEXT) or whether the search starts from the beginning of the list (SQL_FETCH_FIRST).
UCHAR FAR *	*szDriverDesc*	Output	Pointer to storage for the driver description.
SWORD	*cbDriverDescMax*	Input	Maximum length of the *szDriverDesc* buffer.
SWORD FAR *	*pcbDriverDesc*	Output	Total number of bytes (excluding the null-termination byte) available to return in *szDriverDesc*. If the number of bytes available to return is greater than or equal to *cbDriverDescMax*, the driver description in *szDriverDesc* is truncated to *cbDriverDescMax* – 1 bytes.

(1 of 2)

Type	Argument	Use	Description
UCHAR FAR *	*szDriverAttributes*	Output	Pointer to storage for the list of driver attribute value pairs. For more information, see "Usage" on page 12-106.
SWORD	*cbDrvrAttrMax*	Input	Maximum length of the *szDriverAttributes* buffer.
SWORD FAR *	*pcbDrvrAttr*	Output	Total number of bytes (excluding the null-termination byte) available to return in *szDriverAttributes*. If the number of bytes available to return is greater than or equal to *cbDrvrAttrMax*, the list of attribute-value pairs in *szDriverAttributes* is truncated to *cbDrvrAttrMax* – 1 bytes.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	(DM) The buffer *szDriverDesc* was not large enough to return the entire driver description, so the description was truncated. The argument *pcbDriverDesc* contains the length of the entire driver description (function returns SQL_SUCCESS_WITH_INFO). (DM) The buffer *szDriverAttributes* was not large enough to return the entire list of attribute-value pairs, so the list was truncated. The argument *pcbDrvrAttr* contains the length of the untruncated list of attribute-value pairs (function returns SQL_SUCCESS_WITH_INFO).
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	(DM) The driver manager did not allocate memory required to support execution or completion of the function.
S1090	Invalid string or buffer length	(DM) The value specified for argument *cbDriverDescMax* was less than 0. (DM) The value specified for argument *cbDrvrAttrMax* was less than 0 or equal to 1.
S1103	Direction option out of range	(DM) The value specified for the argument *fDirection* was not equal to SQL_FETCH_FIRST or SQL_FETCH_NEXT.

Usage

SQLDrivers returns the driver description in the *szDriverDesc* argument. It returns additional information about the driver in the *szDriverAttributes* argument as a list of keyword-value pairs. Each pair is terminated with a null byte, and the entire list is terminated with a null byte (that is, 2 null bytes mark the end of the list).

If *szDriverAttributes* cannot hold the entire list, the list is truncated, **SQLDrivers** returns SQLSTATE 01004 (Data truncated), and the length of the list (excluding the final null-termination byte) is returned in *pcbDrvrAttr*. Driver-attribute keywords are added during the installation procedure.

An application can call **SQLDrivers** multiple times to retrieve all driver descriptions. A driver manager retrieves this information from the **odbcinst.ini** file. When no more driver descriptions remain, **SQLDrivers** returns SQL_NO_DATA_FOUND. If **SQLDrivers** is called with SQL_FETCH_NEXT immediately after it returns SQL_NO_DATA_FOUND, it returns the first driver description.

If SQL_FETCH_NEXT passes to **SQLDrivers** the first time it is called, **SQLDrivers** returns the first data-source name.

Related Functions

For Information About	See
Discovering and listing values required to connect to a data source	**SQLBrowseConnect**
Connecting to a data source	**SQLConnect**
Returning data source names	**SQLDataSources**
Connecting to a data source using a connection string or dialog box	**SQLDriverConnect**

♦

Core

SQLError

SQLError returns error or status information.

Syntax

```
RETCODE SQLError(henv, hdbc, hstmt, szSqlState,
pfNativeError, szErrorMsg, cbErrorMsgMax, pcbErrorMsg)
```

The **SQLError** function accepts the following arguments.

Type	Argument	Use	Description
HENV	*henv*	Input	Environment handle or SQL_NULL_HENV.
HDBC	*hdbc*	Input	Connection handle or SQL_NULL_HDBC.
HSTMT	*hstmt*	Input	Statement handle or SQL_NULL_HSTMT.
UCHAR FAR *	*szSqlState*	Output	SQLSTATE as null-terminated string. For a list of SQLSTATE values, see Appendix A, "INFORMIX-CLI Error Codes."
SDWORD FAR *	*pfNativeError*	Output	Native error code (specific to the data source).
UCHAR FAR *	*szErrorMsg*	Output	Pointer to storage for the error message text.
SWORD	*cbErrorMsgMax*	Input	Maximum length of the *szErrorMsg* buffer. This value must be less than or equal to SQL_MAX_MESSAGE_LENGTH – 1.
SWORD FAR *	*pcbErrorMsg*	Output	Pointer to the total number of bytes (excluding the null-termination byte) available to return in *szErrorMsg*. If the number of bytes available to return is greater than or equal to *cbErrorMsgMax*, the error message text in *szErrorMsg* is truncated to *cbErrorMsgMax* – 1 bytes.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

SQLError does not post error values for itself. **SQLError** returns SQL_NO_DATA_FOUND when it cannot retrieve any error information (in which case *szSqlState* equals 00000). If **SQLError** cannot access error values for any reason that would normally return SQL_ERROR, **SQLError** returns SQL_ERROR but does not post any error values. If the buffer for the error message is too short, **SQLError** returns SQL_SUCCESS_WITH_INFO but still does not return a SQLSTATE value for **SQLError**.

To determine that a truncation occurred in the error message, an application can compare *cbErrorMsgMax* to the actual length of the message text written to *pcbErrorMsg*.

Usage

An application typically calls **SQLError** when a previous call to an INFORMIX-CLI function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR. However, any INFORMIX-CLI function can post zero or more errors each time that it is called, so an application can call **SQLError** after any INFORMIX-CLI function call.

SQLError retrieves an error from the data structure associated with the right-most non-null handle argument. An application requests error information, in the following ways:

- To retrieve errors associated with an environment, the application passes the corresponding *henv* and includes SQL_NULL_HDBC and SQL_NULL_HSTMT in *hdbc* and *hstmt*, respectively. The driver returns the error status of the INFORMIX-CLI function most recently called with the same *henv*.
- To retrieve errors associated with a connection, the application passes the corresponding *hdbc* plus an *hstmt* equal to SQL_NULL_HSTMT. In such a case, the driver ignores the *henv* argument. The driver returns the error status of the INFORMIX-CLI function most recently called with the *hdbc*.

- To retrieve errors associated with a statement, an application passes the corresponding *hstmt*. If the call to **SQLError** contains a valid *hstmt*, the driver ignores the *hdbc* and *henv* arguments. The driver returns the error status of the INFORMIX-CLI function most recently called with the *hstmt*.
- To retrieve multiple errors for a function call, an application calls **SQLError** multiple times. For each error, the driver returns SQL_SUCCESS and removes that error from the list of available errors.

When no additional information for the right-most non-null handle remains, **SQLError** returns SQL_NO_DATA_FOUND. In this case, *szSqlState* equals 00000 (Success), *pfNativeError* is undefined, *pcbErrorMsg* equals 0, and *szErrorMsg* contains a single null-termination byte (unless *cbErrorMsgMax* equals 0).

The driver manager, if you are using one, stores error information in its *henv*, *hdbc*, and *hstmt* structures. Similarly, the driver stores error information in its *henv*, *hdbc*, and *hstmt* structures. When the application calls **SQLError**, the driver manager checks if any errors exist in its structure for the specified handle. If errors exist for the specified handle, it returns the first error; if no errors exist, it calls **SQLError** in the driver.

The driver manager, if you are using one, can store up to 64 errors with an *henv* and its associated *hdbcs* and *hstmts*. When this limit is reached, the driver manager discards any subsequent errors posted on the *henv*, *hdbcs*, or *hstmts* of the driver manager. The number of errors that a driver can store is driver dependent.

An error is removed from the structure associated with a handle when **SQLError** is called for that handle and returns the error. All errors stored for a specific handle are removed when the handle is used in a subsequent function call. For example, errors on an *hstmt* that **SQLExecDirect** returns are removed when **SQLExecDirect** or **SQLTables** is called with that *hstmt*. The errors stored on a specific handle are not removed as the result of a call to a function that uses an associated handle of a different type. For example, errors on an *hdbc* that **SQLNativeSql** returns are not removed when **SQLError** or **SQLExecDirect** is called with an *hstmt* associated with that *hdbc*.

For more information about error codes, see Appendix A.

Related Functions

None.

♦

Core

SQLExecDirect

SQLExecDirect executes a preparable statement, using the current values of the parameter-marker variables if any parameters exist in the statement. **SQLExecDirect** is the fastest way to submit an SQL statement for one-time execution.

Syntax

```
   RETCODE SQLExecDirect(hstmt, szSqlStr, cbSqlStr)
```

The **SQLExecDirect** function uses the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle
UCHAR FAR *	*szSqlStr*	Input	SQL statement to be executed
SDWORD	*cbSqlStr*	Input	Length of *szSqlStr*

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The argument *szSqlStr* contained an SQL statement that contained a character or binary parameter or literal, and the value exceeded the maximum length of the associated table column. The argument *szSqlStr* contained an SQL statement that contained a numeric parameter or literal, and the fractional part of the value was truncated. The argument *szSqlStr* contained an SQL statement that contained a date or time parameter or literal, and a time-stamp value was truncated.
01006	Privilege not revoked	The argument *szSqlStr* contained a REVOKE statement, and the user did not have the specified privilege (function returns SQL_SUCCESS_WITH_INFO).
01S03	No rows updated or deleted	The argument *szSqlStr* contained a positioned UPDATE or DELETE statement, and no rows were updated or deleted (function returns SQL_SUCCESS_WITH_INFO).
01S04	More than one row updated or deleted	The argument *szSqlStr* contained a positioned UPDATE or DELETE statement, and more than one row was updated or deleted (function returns SQL_SUCCESS_WITH_INFO).
07001	Wrong number of parameters	The number of parameters specified in **SQLBindParameter** was less than the number of parameters in the SQL statement contained in the argument *szSqlStr*.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.

(1 of 5)

SQLSTATE	Error	Description
21S01	Insert value list does not match column list	The argument *szSqlStr* contained an INSERT statement, and the number of values to be inserted did not match the degree of the derived table.
21S02	Degree of derived table does not match column list	The argument *szSqlStr* contained a CREATE VIEW statement, and the number of names specified is not the same degree as the derived table defined by the query specification.
22003	Numeric value out of range	The argument *szSqlStr* contained an SQL statement that contained a numeric parameter or literal, and the value caused the whole (as opposed to fractional) part of the number to be truncated when assigned to the associated table column.
22005	Error in assignment	The argument *szSqlStr* contained an SQL statement that contained a parameter or literal, and the value was incompatible with the data type of the associated table column.
22008	Datetime field overflow	The argument *szSqlStr* contained an SQL statement that contained a date, time, or time-stamp parameter or literal, and the value was, respectively, an invalid date, time, or time stamp.
22012	Division by zero	The argument *szSqlStr* contained an SQL statement that contained an arithmetic expression that caused division by zero.
23000	Integrity-constraint violation	The argument *szSqlStr* contained an SQL statement that contained a parameter or literal. The parameter value was NULL for a column defined as NOT NULL in the associated table column, a duplicate value was supplied for a column constrained to contain only unique values, or some other integrity constraint was violated.
24000	Invalid cursor state	A cursor was already opened on the statement handle. The argument *szSqlStr* contained a positioned UPDATE or DELETE statement, but the cursor was positioned before the start of the result set or after the end of the result set.
34000	Invalid cursor name	The argument *szSqlStr* contained a positioned UPDATE or DELETE statement, but the cursor referenced by the statement being executed was not open.
37000	Syntax error or access violation	The argument *szSqlStr* contained an SQL statement that was not preparable or contained a syntax error.
40001	Serialization failure	The transaction to which the SQL statement contained in the argument *szSqlStr* belonged was terminated to prevent deadlock.

(2 of 5)

SQLSTATE	Error	Description
42000	Syntax error or access violation	The user did not have permission to execute the SQL statement contained in the argument *szSqlStr*.
S0001	Base table or view already exists	The argument *szSqlStr* contained a CREATE TABLE or CREATE VIEW statement, but the table name or view name specified already exists.
S0002	Table or view not found	The argument *szSqlStr* contained a DROP TABLE or a DROP VIEW statement, but the specified table name or view name did not exist. The argument *szSqlStr* contained an ALTER TABLE statement, but the specified table name did not exist. The argument *szSqlStr* contained a CREATE VIEW statement, but a table name or view name defined by the query specification did not exist. The argument *szSqlStr* contained a CREATE INDEX statement, but the specified table name did not exist. The argument *szSqlStr* contained a GRANT or REVOKE statement, but the specified table name or view name did not exist. The argument *szSqlStr* contained a SELECT statement, but a specified table name or view name did not exist. The argument *szSqlStr* contained a DELETE, INSERT, or UPDATE statement, but the specified table name did not exist. The argument *szSqlStr* contained a CREATE TABLE statement, but a table specified in a constraint (referencing a table other than the one being created) did not exist.
S0011	Index already exists	The argument *szSqlStr* contained a CREATE INDEX statement, but the specified index name already existed.
S0012	Index not found	The argument *szSqlStr* contained a DROP INDEX statement, but the specified index name did not exist.
S0021	Column already exists	The argument *szSqlStr* contained an ALTER TABLE statement, but the column specified in the ADD clause is not unique or identifies an existing column in the base table.

(3 of 5)

SQLSTATE	Error	Description
S0022	Column not found	The argument *szSqlStr* contained a CREATE INDEX statement, but one or more of the column names specified in the column list did not exist. The argument *szSqlStr* contained a GRANT or REVOKE statement, but a specified column name did not exist. The argument *szSqlStr* contained a SELECT, DELETE, INSERT, or UPDATE statement, but a specified column name did not exist. The argument *szSqlStr* contained a CREATE TABLE statement, but a column specified in a constraint (referencing a table other than the one being created) did not exist.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1009	Invalid argument value	(DM) The argument *szSqlStr* was a null pointer.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The argument *cbSqlStr* was less than or equal to 0, but not equal to SQL_NTS. A parameter value, set with **SQLBindParameter**, was a null pointer and the parameter length value was not 0, SQL_NULL_DATA, SQL_DATA_AT_EXEC, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. A parameter value, set with **SQLBindParameter**, was not a null pointer and the parameter length value was less than 0, but was not SQL_NTS, SQL_NULL_DATA, SQL_DATA_AT_EXEC, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.

(4 of 5)

SQLSTATE	Error	Description
S1109	Invalid cursor position	The argument *szSqlStr* contained a positioned UPDATE or DELETE statement, but the cursor was positioned (by **SQLExtendedFetch**) on a row for which the value in the *rgfRowStatus* array in **SQLExtendedFetch** was SQL_ROW_DELETED or SQL_ROW_ERROR.
S1C00	Driver not capable	The combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options was not supported by the driver or data source.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(5 of 5)

Usage

The application calls **SQLExecDirect** to send an SQL statement to the data source. The driver modifies the statement to use the form of SQL that the data source uses and then submits it to the data source. In particular, the driver modifies the escape clauses used to define ODBC-specific SQL.

The application can include one or more parameter markers in the SQL statement. To include a parameter marker, the application embeds a question mark (?) into the SQL statement at the appropriate position.

If the SQL statement is a SELECT statement, and if the application called **SQLSetCursorName** to associate a cursor with an *hstmt*, the driver uses the specified cursor. Otherwise, the driver generates a cursor name.

If the data source is in manual-commit mode (requiring explicit transaction initiation), and a transaction has not been initiated, the driver initiates a transaction before it sends the SQL statement.

If an application uses **SQLExecDirect** to submit a COMMIT or ROLLBACK statement, it is not interoperable between DBMS products. To commit or roll back a transaction, call **SQLTransact**.

If **SQLExecDirect** encounters a data-at-execution parameter, it returns SQL_NEED_DATA. The application sends the data using **SQLParamData** and **SQLPutData**. For more information, see **SQLBindParameter**, **SQLParamData**, and **SQLPutData**.

Code Example

See **SQLBindCol**, **SQLExtendedFetch**, **SQLProcedures**, and **SQLGetData**.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Executing a prepared SQL statement	**SQLExecute**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning a cursor name	**SQLGetCursorName**
Fetching part or all of a column of data	**SQLGetData**
Returning the next parameter to send data for	**SQLParamData**
Preparing a statement for execution	**SQLPrepare**
Sending parameter data at execution time	**SQLPutData**
Setting a cursor name	**SQLSetCursorName**
Setting a statement option	**SQLSetStmtOption**
Executing a commit or rollback operation	**SQLTransact**

♦

Core

SQLExecute

SQLExecute executes a prepared statement, using the current values of the parameter-marker variables if any parameter markers exist in the statement.

Syntax

```
RETCODE SQLExecute(hstmt)
```

The **SQLExecute** statement accepts the following argument.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The prepared statement associated with the *hstmt* contained a character or binary parameter or literal, but the value exceeded the maximum length of the associated table column. The prepared statement associated with the *hstmt* contained a numeric parameter or literal, but the fractional part of the value was truncated. The prepared statement associated with the *hstmt* contained a date or time parameter or literal, and a time-stamp value was truncated.
01006	Privilege not revoked	The prepared statement associated with the *hstmt* was REVOKE, but the user did not have the specified privilege (function returns SQL_SUCCESS_WITH_INFO).
01S03	No rows updated or deleted	The prepared statement associated with the *hstmt* was a positioned UPDATE or DELETE statement, but no rows were updated or deleted (function returns SQL_SUCCESS_WITH_INFO).
01S04	More than one row updated or deleted	The prepared statement associated with the *hstmt* was a positioned UPDATE or DELETE statement, and more than one row was updated or deleted (function returns SQL_SUCCESS_WITH_INFO).
07001	Wrong number of parameters	The number of parameters specified in **SQLBindParameter** was less than the number of parameters in the prepared statement associated with the *hstmt*.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.

(1 of 4)

SQLSTATE	Error	Description
22003	Numeric value out of range	The prepared statement associated with the *hstmt* contained a numeric parameter, and the parameter value caused the whole (as opposed to fractional) part of the number to be truncated when assigned to the associated table column.
22005	Error in assignment	The prepared statement associated with the *hstmt* contained a parameter, but the value was incompatible with the data type of the associated table column.
22008	Datetime field overflow	The prepared statement associated with the *hstmt* contained a date, time, or time-stamp parameter or literal, and the value was an invalid date, time, or time stamp, respectively.
22012	Division by zero	The prepared statement associated with the *hstmt* contained an arithmetic expression that caused division by zero.
23000	Integrity constraint violation	The prepared statement associated with the *hstmt* contained a parameter. The parameter value was NULL for a column defined as NOT NULL in the associated table column, a duplicate value was supplied for a column constrained to contain only unique values, or some other integrity constraint was violated.
24000	Invalid cursor state	A cursor was already opened on the statement handle. The prepared statement associated with the *hstmt* contained a positioned update or delete statement, and the cursor was positioned before the start of the result set or after the end of the result set.
40001	Serialization failure	The transaction to which the prepared statement associated with the *hstmt* belonged was terminated to prevent deadlock.
42000	Syntax error or access violation	The user did not have permission to execute the prepared statement associated with the *hstmt*.

(2 of 4)

SQLSTATE	Error	Description
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support executing or completing the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
		(DM) The *hstmt* was not prepared. Either the *hstmt* was not in an executed state, or a cursor was open on the *hstmt* and **SQLFetch** or **SQLExtendedFetch** had been called.
		The *hstmt* was not prepared. It was in an executed state, and either no result set was associated with the *hstmt* or **SQLFetch** or **SQLExtendedFetch** had not been called.
S1090	Invalid string or buffer length	A parameter value, set with **SQLBindParameter**, was a null pointer, and the parameter length value was not 0, SQL_NULL_DATA, SQL_DATA_AT_EXEC, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.
		A parameter value, set with **SQLBindParameter**, was not a null pointer, and the parameter length value was less than 0, but was not SQL_NTS, SQL_NULL_DATA, or SQL_DATA_AT_EXEC, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.

(3 of 4)

SQLSTATE	Error	Description
S1109	Invalid cursor position	The prepared statement was a positioned UPDATE or DELETE statement, and the cursor was positioned (by **SQLExtendedFetch**) on a row for which the value in the *rgfRowStatus* array in **SQLExtendedFetch** was SQL_ROW_DELETED or SQL_ROW_ERROR.
S1C00	Driver not capable	The combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options was not supported by the driver or data source.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(4 of 4)

SQLExecute can return any SQLSTATE that **SQLPrepare** can return based on when the data source evaluates the SQL statement associated with the *hstmt*.

Usage

SQLExecute executes a statement prepared by **SQLPrepare**. Once the application processes or discards the results from a call to **SQLExecute**, the application can call **SQLExecute** again with new parameter values.

To execute a SELECT statement more than once, the application must call **SQLFreeStmt** with the SQL_CLOSE parameter before it reissues the SELECT statement.

If the data source is in manual-commit mode (requiring explicit transaction initiation), and a transaction has not already been initiated, the driver initiates a transaction before it sends the SQL statement.

If an application uses **SQLPrepare** to prepare and **SQLExecute** to submit a COMMIT or ROLLBACK statement, it is not interoperable between DBMS products. To commit or roll back a transaction, call **SQLTransact**.

If **SQLExecute** encounters a data-at-execution parameter, it returns SQL_NEED_DATA. The application sends the data using **SQLParamData** and **SQLPutData**. For more information, see **SQLBindParameter**, **SQLParamData**, and **SQLPutData**.

Code Example

See **SQLBindParameter** and **SQLPutData**.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Executing an SQL statement	**SQLExecDirect**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Freeing a statement handle	**SQLFreeStmt**
Returning a cursor name	**SQLGetCursorName**
Fetching part or all of a column of data	**SQLGetData**
Returning the next parameter to send data for	**SQLParamData**
Preparing a statement for execution	**SQLPrepare**
Sending parameter data at execution time	**SQLPutData**
Setting a cursor name	**SQLSetCursorName**
Setting a statement option	**SQLSetStmtOption**
Executing a commit or rollback operation	**SQLTransact**

♦

Level 2

SQLExtendedFetch

SQLExtendedFetch extends the functionality of **SQLFetch** in the following ways:

- It returns row-set data (one or more rows), in the form of an array, for each bound column.
- It scrolls through the result set according to the setting of a scroll-type argument.

SQLExtendedFetch works with **SQLSetStmtOption**.

To fetch one row of data at a time in a forward direction, an application should call **SQLFetch**.

For more information about scrolling through result sets, see "Using Block and Scrollable Cursors" on page 6-9.

Syntax

```
RETCODE SQLExtendedFetch(hstmt, fFetchType, irow, pcrow,
rgfRowStatus)
```

The **SQLExtendedFetch** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle
UWORD	*fFetchType*	Input	Type of fetch
SDWORD	*irow*	Input	Number of the row to fetch
UDWORD FAR *	*pcrow*	Output	Number of rows actually fetched
UWORD FAR *	*rgfRowStatus*	Output	An array of status values

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The data returned for one or more columns was truncated. String values are right truncated. For numeric values, the fractional part of number was truncated (function returns SQL_SUCCESS_WITH_INFO).
01S01	Error in row	An error occurred while fetching one or more rows (function returns SQL_SUCCESS_WITH_INFO).
07006	Restricted data type attribute violation	A data value could not be converted to the C data type specified by *fCType* in **SQLBindCol**.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
22002	Indicator value required but not supplied	If the column value for any bound column is null, the INDICATOR_PTR field of the corresponding descriptor record must not be a null pointer.

(1 of 3)

SQLSTATE	Error	Description
22003	Numeric value out of range	Returning the numeric value (as numeric or string) for one or more columns would cause the whole (as opposed to fractional) part of the number to be truncated. Returning the binary value for one or more columns would cause a loss of binary significance.
22005	Error in assignment	A zero-length string was inserted into a string field, and the string was bound to a numeric data type, so the string was converted to a zero.
22008	Datetime field overflow	An SQL_C_TIME, SQL_C_DATE, or SQL_C_TIMESTAMP value was converted to an SQL_CHAR data type, and the value was an invalid date, time, or time stamp, respectively.
22012	Division by zero	A value from an arithmetic expression was returned which resulted in division by zero.
24000	Invalid cursor state	The *hstmt* was in an executed state, but no result set was associated with the *hstmt*.
40001	Serialization failure	The transaction (in which the fetch was executed) was terminated to prevent deadlock.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1002	Invalid column number	A column number specified in the binding for one or more columns was greater than the number of columns in the result set. Column 0 was bound with **SQLBindCol** and the SQL_USE_BOOKMARKS statement option was set to SQL_UB_OFF.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.

(2 of 3)

SQLSTATE	Error	Description
S1010	Function-sequence error	(DM) The specified *hstmt* was not in an executed state. The function was called without first calling **SQLExecDirect**, **SQLExecute**, or a catalog function. (DM) **SQLExecute** or **SQLExecDirect**, was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) **SQLExtendedFetch** was called for an *hstmt* after **SQLFetch** was called and before **SQLFreeStmt** was called with the SQL_CLOSE option.
S1106	Fetch type out of range	(DM) The value specified for the argument *fFetchType* was invalid. The value of the SQL_CURSOR_TYPE statement option was SQL_CURSOR_FORWARD_ONLY and the value of argument *fFetchType* was not SQL_FETCH_NEXT.
S1107	Row value out of range	The value specified with the statement option SQL_CURSOR_TYPE was SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the SQL_KEYSET_SIZE statement option was greater than 0 and less than the value specified with the SQL_ROWSET_SIZE statement option.
S1C00	Driver not capable	The driver or data source does not support the specified fetch type. The driver or data source does not support the conversion specified by the combination of the *fCType* in **SQLBindCol** and the SQL data type of the corresponding column. This error applies only when the SQL data type of the column was mapped to a driver-specific SQL data type. The argument *fFetchType* was SQL_FETCH_RESUME, and the driver supports ODBC 2.0.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(3 of 3)

Usage

SQLExtendedFetch returns one row set of data to the application. An application cannot mix calls to **SQLExtendedFetch** and **SQLFetch** for the same cursor.

An application specifies the number of rows in the row set by calling **SQLSetStmtOption** with the SQL_ROWSET_SIZE statement option.

Binding

If any columns in the result set are bound with **SQLBindCol**, the driver converts the data for the bound columns as necessary and stores it in the locations bound to those columns. The result set can be bound in a column-wise (the default) or row-wise fashion.

Column-Wise Binding

To bind a result set in column-wise fashion, an application specifies SQL_BIND_BY_COLUMN for the SQL_BIND_TYPE statement option.

To bind each column

1. The application allocates an array of data-storage buffers. The array has as many elements as the row set has rows, plus an additional element if the application searches for key values or appends new rows of data. The size of each buffer is the maximum size of the C data that can be returned for the column. For example, when the C data type is SQL_C_DEFAULT, the size of each buffer is the column length. When the C data type is SQL_C_CHAR, the size of each buffer is the display size of the data. For more information, see "Converting Data from SQL to C" on page B-19 and"Precision, Scale, Length, and Display Size" on page B-5.
2. The application allocates an array of SDWORDs to hold the number of bytes available to return for each row in the column. The array has as many elements as the row set has rows.
3. The application calls **SQLBindCol**:
   - The *rgbValue* argument specifies the address of the data-storage array.
   - The *cbValueMax* argument specifies the size of each buffer in the data-storage array.
   - The *pcbValue* argument specifies the address of the number-of-bytes array.

When the application calls **SQLExtendedFetch**, the driver retrieves the data and the number of bytes that are available to return and stores them in the buffers that the application allocates, as the following actions show:

- For each bound column, the driver stores the data in the *rgbValue* buffer bound to the column. It stores the first row of data at the start of the buffer and each subsequent row of data at an offset of *cbValueMax* bytes from the data for the previous row.
- For each bound column, the driver stores the number of bytes that are available to return in the *pcbValue* buffer that is bound to the column. This is the number of bytes available before calling **SQLExtendedFetch**. (If the number of bytes available to return cannot be determined in advance, the driver sets *pcbValue* to SQL_NO_TOTAL. If the data for the column is NULL, the driver sets *pcbValue* to SQL_NULL_DATA.) It stores the number of bytes available to return for the first row at the start of the buffer and the number of bytes available to return for each subsequent row at an offset of `sizeof(SDWORD)` from the value for the previous row.

Row-Wise Binding

To bind a result set in row-wise fashion, an application must specify the SQL_BIND_TYPE statement option for **SQLSetStmtOption**.

To bind a result set in row-wise fashion

1. The application declares a structure that can hold a single row of retrieved data and the associated data lengths. For each bound column, the structure contains one field for the data and one SDWORD field for the number of bytes that are available to return. The size of the data field is the maximum size of the C data that can be returned for the column.
2. The application calls **SQLSetStmtOption** with *fOption* set to SQL_BIND_TYPE and *vParam* set to the size of the structure.
3. The application allocates an array of these structures. The array has as many elements as the row set has rows, plus an additional element if the application searches for key values or appends new rows of data.

4. The application calls **SQLBindCol** for each column to be bound:
   - The *rgbValue* argument specifies the address of the data field of the column in the first array element.
   - The *cbValueMax* argument specifies the size of the data field of the column.
   - The *pcbValue* argument specifies the address of the number-of-bytes field of the column in the first array element.

When the application calls **SQLExtendedFetch**, the driver retrieves the data and the number of bytes that are available to return and stores them in the buffers that are allocated by the application:

- For each bound column, the driver stores the first row of data at the address specified by *rgbValue* for the column and each subsequent row of data at an offset of *vParam* bytes from the data for the previous row.
- For each bound column, the driver stores the number of bytes that are available to return for the first row at the address that *pcbValue* specifies and the number of bytes that are available to return for each subsequent row at an offset of *vParam* bytes from the value for the previous row. This is the number of bytes that are available prior to calling **SQLExtendedFetch**. (If the number of bytes that are available to return cannot be determined in advance, the driver sets *pcbValue* to SQL_NO_TOTAL. If the data for the column is NULL, the driver sets *pcbValue* to SQL_NULL_DATA.)

Positioning the Cursor

The following operations require a cursor position:

- Positioned update and delete statements
- Calls to **SQLGetData**

Before the application executes a positioned update, a delete statement, or a call to **SQLGetData**, it must position the cursor by calling **SQLExtendedFetch** to retrieve a row set; the cursor points to the first row in the row set.

The following table shows the row set and code returned when the application requests different row sets.

Requested Row Set	Return Code	Cursor Position	Returned Row Set
Before start of result set	SQL_NO_DATA_FOUND	Before start of result set	None. The contents of the row-set buffers are undefined.
Overlaps start of result set	SQL_SUCCESS	Row 1 of row set	First row set in result set.
Within result set	SQL_SUCCESS	Row 1 of row set	Requested row set.
Overlaps end of result set	SQL_SUCCESS	Row 1 of row set	For rows in the row set that overlap the result set, data is returned. For rows in the row set outside the result set, the contents of the *rgbValue* and *pcbValue* buffers are undefined, and the *rgfRowStatus* array contains SQL_ROW_NOROW.
After end of result set	SQL_NO_DATA_FOUND	After end of result set	None. The contents of the row-set buffers are undefined.

For example, suppose a result set has 100 rows, and the row-set size is 5. The following table shows the row set and code returned by **SQLExtendedFetch** for different values of *irow* when the fetch type is SQL_FETCH_RELATIVE.

Current Row Set	irow	Return Code	New Row Set
1 to 5	–5	SQL_NO_DATA_FOUND	None.
1 to 5	–3	SQL_SUCCESS	1 to 5.
96 to 100	5	SQL_NO_DATA_FOUND	None.
96 to 100	3	SQL_SUCCESS	99 and 100. For rows 3, 4, and 5 in the row set, the *rgfRowStatusArray* is set to SQL_ROW_NOROW.

Before **SQLExtendedFetch** is called the first time, the cursor is positioned before the start of the result set.

For the purpose of moving the cursor, deleted rows (that is, rows with an entry in the *rgfRowStatus* array of SQL_ROW_DELETED) are treated no differently than other rows. For example, calling **SQLExtendedFetch** with *fFetchType* set to SQL_FETCH_ABSOLUTE and *irow* set to 15 returns the row set starting at row 15, even if the *rgfRowStatus* array for row 15 is SQL_ROW_DELETED.

Processing Errors

If an error occurs that pertains to the entire row set, such as SQLSTATE S1T00 (Time-out expired), the driver returns SQL_ERROR and the appropriate SQLSTATE. The contents of the row set buffers are undefined, and the cursor position is unchanged.

If an error occurs that pertains to a single row, the driver performs the following actions:

- Sets the element in the *rgfRowStatus* array for the row to SQL_ROW_ERROR
- Posts SQLSTATE 01S01 (Error in row) in the error queue
- Posts zero or more additional SQLSTATE values for the error after SQLSTATE 01S01 (Error in row) in the error queue

After the driver processes the error or warning, it continues the operation for the remaining rows in the row set and returns SQL_SUCCESS_WITH_INFO. Thus, for each error that pertains to a single row, the error queue contains SQLSTATE 01S01 (Error in row) followed by zero or more additional SQLSTATEs.

After the driver processes the error, it fetches the remaining rows in the row set and returns SQL_SUCCESS_WITH_INFO. Thus, for each row that returns an error, the error queue contains SQLSTATE 01S01 (Error in row) followed by zero or more additional SQLSTATE values.

If the row set contains rows that are already fetched, the driver is not required to return SQLSTATE values for errors that occurred when the rows were first fetched. However, it is required to return SQLSTATE 01S01 (Error in row) for each row in which an error originally occurred and to return SQL_SUCCESS_WITH_INFO. For example, a static cursor that maintains a cache might cache row-status information (so that it can determine which rows contain errors) but might not cache the SQLSTATE associated with those errors.

Error rows do not affect relative cursor movements. For example, suppose the result set size is 100, and the row-set size is 10. If the current row set is rows 11 through 20 and the element in the *rgfRowStatus* array for row 11 is SQL_ROW_ERROR, calling **SQLExtendedFetch** with the SQL_FETCH_NEXT fetch type still returns rows 21 through 30.

If the driver returns any warnings, such as SQLSTATE 01004 (Data truncated), it returns warnings that apply to the entire row set or to unknown rows in the row set before it returns error information that applies to specific rows. It returns warnings for specific rows along with any other error information about those rows.

fFetchType Argument

The *fFetchType* argument specifies how to move through the result set. It is one of the following values:

- SQL_FETCH_NEXT
- SQL_FETCH_FIRST
- SQL_FETCH_LAST
- SQL_FETCH_PRIOR
- SQL_FETCH_ABSOLUTE
- SQL_FETCH_RELATIVE

If the value of the SQL_CURSOR_TYPE statement option is SQL_CURSOR_FORWARD_ONLY, the *fFetchType* argument must be SQL_FETCH_NEXT.

Tip: *SQL_CURSOR_FORWARD_ONLY is the only option unless an application uses the Microsoft ODBC Cursor Library.*

Moving by Row Position

SQLExtendedFetch supports the following values of the *fFetchType* argument to move the cursor relative to the current row set.

fFetchType Argument	Action
SQL_FETCH_NEXT	The driver returns the next row set. If the cursor is positioned before the start of the result set, this is equivalent to SQL_FETCH_FIRST.
SQL_FETCH_PRIOR	The driver returns the prior row set. If the cursor is positioned after the end of the result set, this is equivalent to SQL_FETCH_LAST.
SQL_FETCH_RELATIVE	The driver returns the row set *irow* rows from the start of the current row set. If *irow* equals 0, the driver refreshes the current row set. If the cursor is positioned before the start of the result set and *irow* is greater than 0, or if the cursor is positioned after the end of the result set and *irow* is less than 0, this is equivalent to SQL_FETCH_ABSOLUTE.

It supports the following values of the *fFetchType* argument to move the cursor to an absolute position in the result set.

fFetchType Argument	Action
SQL_FETCH_FIRST	The driver returns the first row set in the result set.
SQL_FETCH_LAST	The driver returns the last complete row set in the result set.
SQL_FETCH_ABSOLUTE	If *irow* is greater than 0, the driver returns the row set starting at row *irow*. If *irow* equals 0, the driver returns SQL_NO_DATA_FOUND, and the cursor is positioned before the start of the result set. If *irow* is less than 0, the driver returns the row set starting at row *n*+*irow*+1, where *n* is the number of rows in the result set. For example, if *irow* is -1, the driver returns the row set that starts at the last row in the result set. If the result set size is 10 and *irow* is -10, the driver returns the row set that starts at the first row in the result set.

irow Argument

For the SQL_FETCH_ABSOLUTE fetch type, **SQLExtendedFetch** returns the row set that starts at the row number specified by the *irow* argument.

For the SQL_FETCH_RELATIVE fetch type, **SQLExtendedFetch** returns the row set that starts *irow* rows from the first row in the current row set.

The *irow* argument is ignored for the SQL_FETCH_NEXT, SQL_FETCH_PRIOR, SQL_FETCH_FIRST, and SQL_FETCH_LAST fetch types.

rgfRowStatus Argument

In the *rgfRowStatus* array, **SQLExtendedFetch** returns any changes in status to each row since it was last retrieved from the data source. Rows might be unchanged (SQL_ROW_SUCCESS), updated (SQL_ROW_UPDATED), deleted (SQL_ROW_DELETED), added (SQL_ROW_ADDED), or unretrievable due to an error (SQL_ROW_ERROR).

Important: *The Informix driver cannot detect changes to data. The number of elements must equal the number of rows in the row set (as the SQL_ROWSET_SIZE statement option defines). If the number of rows fetched is less than the number of elements in the status array, the driver sets remaining status elements to SQL_ROW_NOROW.*

Code Example

The following two examples show how an application could use column-wise or row-wise binding to bind storage locations to the same result set.

Column-Wise Binding

In the following example, an application declares storage locations for column-wise bound data and the returned numbers of bytes. Because column-wise binding is the default, it is unnecessary to request column-wise binding with **SQLSetStmtOption**, as in the row-wise binding example. However, the application does call **SQLSetStmtOption** to specify the number of rows in the row set.

The application then executes a SELECT statement to return a result set of the employee names and birthdays, which is sorted by birthday. It calls **SQLBindCol** to bind the columns of data, passing the addresses of storage locations for both the data and the returned numbers of bytes. Finally, the application fetches the row-set data with **SQLExtendedFetch** and prints each employee's name and birthday.

```
#define ROWS 100
#define NAME_LEN 30
#define BDAY_LEN 11

UCHAR     szName[ROWS][NAME_LEN], szBirthday[ROWS][BDAY_LEN];
SWORD     sAge[ROWS];
SDWORD    cbName[ROWS], cbAge[ROWS], cbBirthday[ROWS];

UDWORD    crow, irow;
UWORD     rgfRowStatus[ROWS];

SQLSetStmtOption(hstmt, SQL_CONCURRENCY, SQL_CONCUR_READ_ONLY);
SQLSetStmtOption(hstmt, SQL_CURSOR_TYPE, SQL_CURSOR_KEYSET_DRIVEN);
SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWS);
retcode = SQLExecDirect(hstmt,

            "SELECT NAME, AGE, BIRTHDAY FROM EMPLOYEE ORDER BY 3, 2, 1",
            SQL_NTS);

if (retcode = = SQL_SUCCESS) {
    SQLBindCol(hstmt, 1, SQL_C_CHAR, szName, NAME_LEN, cbName);
    SQLBindCol(hstmt, 2, SQL_C_SSHORT, sAge, 0, cbAge);
    SQLBindCol(hstmt, 3, SQL_C_CHAR, szBirthday, BDAY_LEN,
               cbBirthday);

    /* Fetch the rowset data and print each row. */
    /* On an error, display a message and exit.  */

    while (TRUE) {
        retcode = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 1, &crow,
                                   rgfRowStatus);
        if (retcode = = SQL_ERROR || retcode = = SQL_SUCCESS_WITH_INFO) {
            show_error();
        }
        if (retcode = = SQL_SUCCESS || retcode = = SQL_SUCCESS_WITH_INFO){
            for (irow = 0; irow < crow; irow++) {
                if (rgfRowStatus[irow] != SQL_ROW_DELETED &&

                    rgfRowStatus[irow] != SQL_ROW_ERROR)
                    fprintf(out, "%-*s  %-2d  %*s",

                            NAME_LEN-1, szName[irow], sAge[irow],

                            BDAY_LEN-1, szBirthday[irow]);
            }
        } else {
            break;
        }
    }
}
```

Row-Wise Binding

In the following example, an application declares an array of structures to hold row-wise bound data and the returned numbers of bytes. Using **SQLSetStmtOption**, the application requests row-wise binding and passes the size of the structure to the driver. The driver uses this size to find successive storage locations in the array of structures and specifies the size of the row set with **SQLSetStmtOption**.

The application then executes a SELECT statement to return a result set of the employee names and birthdays, which is sorted by birthday. It calls **SQLBindCol** to bind the columns of data, passing the addresses of storage locations for both the data and the returned numbers of bytes. Finally, the application fetches the row-set data with **SQLExtendedFetch** and prints each employee's name and birthday.

```
#define ROWS 100
#define NAME_LEN 30
#define BDAY_LEN 11

typedef struct {
    UCHAR     szName[NAME_LEN];
    SDWORD    cbName;
    SWORD     sAge;
    SDWORD    cbAge;
    UCHAR     szBirthday[BDAY_LEN];
    SDWORD    cbBirthday;
    }   EmpTable;

EmpTable rget[ROWS];
UDWORD   crow, irow;
UWORD    rgfRowStatus[ROWS];

SQLSetStmtOption(hstmt, SQL_BIND_TYPE, sizeof(EmpTable));
SQLSetStmtOption(hstmt, SQL_CONCURRENCY, SQL_CONCUR_READ_ONLY);
SQLSetStmtOption(hstmt, SQL_CURSOR_TYPE, SQL_CURSOR_KEYSET_DRIVEN);
SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWS);
retcode = SQLExecDirect(hstmt,
            "SELECT NAME, AGE, BIRTHDAY FROM EMPLOYEE ORDER BY 3, 2, 1",
            SQL_NTS);

if (retcode = = SQL_SUCCESS) {
    SQLBindCol(hstmt, 1, SQL_C_CHAR, rget[0].szName, NAME_LEN,
               &rget[0].cbName);
    SQLBindCol(hstmt, 2, SQL_C_SSHORT, &rget[0].sAge, 0,
              &rget[0].cbAge);
    SQLBindCol(hstmt, 3, SQL_C_CHAR, rget[0].szBirthday, BDAY_LEN,
              &rget[0].cbBirthday);

    /* Fetch the rowset data and print each row. */
    /* On an error, display a message and exit.  */

    while (TRUE) {
        retcode = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 1, &crow,
                                   rgfRowStatus);
        if (retcode = = SQL_ERROR || retcode = = SQL_SUCCESS_WITH_INFO) {
            show_error();
        }
        if (retcode = = SQL_SUCCESS || retcode = = SQL_SUCCESS_WITH_INFO){
            for (irow = 0; irow < crow; irow++) {
                if (rgfRowStatus[irow] != SQL_ROW_DELETED &&
                    rgfRowStatus[irow] != SQL_ROW_ERROR)
                    fprintf(out, "%-*s  %-2d  %*s",
                         NAME_LEN-1, rget[irow].szName, rget[irow].sAge,
                         BDAY_LEN-1, rget[irow].szBirthday);
            }
        } else {
            break;
        }
    }
}
```

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning information about a column in a result set	**SQLDescribeCol**
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Returning the number of result set columns	**SQLNumResultCols**
Setting a statement option	**SQLSetStmtOption**

♦

Core

SQLFetch

SQLFetch fetches a row of data from a result set. The driver returns data for all columns that were bound to storage locations with **SQLBindCol**.

Syntax

```
RETCODE SQLFetch(hstmt)
```

The **SQLFetch** function accepts the following argument.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The data returned for one or more columns was truncated. String values are right truncated. For numeric values, the fractional part of number was truncated (function returns SQL_SUCCESS_WITH_INFO).
07006	Restricted data-type attribute violation	The data value could not be converted to the data type specified by *fCType* in **SQLBindCol**.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
22002	Indicator value required but not supplied	If the column value for any bound column is null, the INDICATOR_PTR field of the corresponding descriptor record must not be a null pointer.
22003	Numeric value out of range	Returning the numeric value (as numeric or string) for one or more columns would have caused the whole (as opposed to fractional) number to truncate. Returning the binary value for one or more columns would cause a loss of binary significance. For more information, see "Converting Data from SQL to C" on page B-19.

(1 of 3)

SQLSTATE	Error	Description
22005	Error in assignment	A zero-length string was inserted into a string field, and the string was bound to a numeric data type, so the string was converted to a zero.
22008	Datetime field overflow	An SQL_C_TIME, SQL_C_DATE, or SQL_C_TIMESTAMP value was converted to an SQL_CHAR data type, and the value was an invalid date, time, or time stamp, respectively.
22012	Division by zero	A value from an arithmetic expression was returned that resulted in division by zero.
24000	Invalid cursor state	The *hstmt* was in an executed state, but no result set was associated with the *hstmt*.
40001	Serialization failure	The transaction in which the fetch was executed was terminated to prevent deadlock.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support executing or completing the function.
S1002	Invalid column number	A column number specified in the binding for one or more columns was greater than the number of columns in the result set. A column number specified in the binding for a column was 0.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.

(2 of 3)

SQLSTATE	Error	Description
S1010	Function-sequence error	(DM) The specified *hstmt* was not in an executed state. The function was called without first calling **SQLExecDirect**, **SQLExecute**, or a catalog function. (DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) **SQLExtendedFetch** was called for an *hstmt* after **SQLFetch** was called and before **SQLFreeStmt** was called with the SQL_CLOSE option.
S1C00	Driver not capable	The driver or data source does not support the conversion specified by the combination of the *fCType* in **SQLBindCol** and the SQL data type of the corresponding column. This error applies only when the SQL data type of the column was mapped to a driver-specific SQL data type.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(3 of 3)

Usage

SQLFetch positions the cursor on the next row of the result set. Before **SQLFetch** is called the first time, the cursor is positioned before the start of the result set. When the cursor is positioned on the last row of the result set, **SQLFetch** returns SQL_NO_DATA_FOUND, and the cursor is positioned after the end of the result set. An application cannot mix calls to **SQLExtendedFetch** and **SQLFetch** for the same cursor.

If the application called **SQLBindCol** to bind columns, **SQLFetch** stores data in the locations specified by the calls to **SQLBindCol**. If the application does not call **SQLBindCol** to bind any columns, **SQLFetch** does not return any data; it moves the cursor to the next row. An application can call **SQLGetData** to retrieve data that is not bound to a storage location.

The driver manages cursors during the fetch operation and places each value of a bound column into the associated storage. The driver follows these guidelines when it performs a fetch operation:

- **SQLFetch** accesses column data in left-to-right order.
- After each fetch, *pcbValue* (specified in **SQLBindCol**) contains the number of bytes that are available to return for the column which is the number of bytes that are available before calling **SQLFetch**. If the number of bytes that are available to return cannot be determined in advance, the driver sets *pcbValue* to SQL_NO_TOTAL. (If SQL_MAX_LENGTH was specified with **SQLSetStmtOption** and the number of bytes that are available to return is greater than SQL_MAX_LENGTH, *pcbValue* contains SQL_MAX_LENGTH.)
- If *rgbValue* cannot hold the entire result, the driver stores part of the value and returns SQL_SUCCESS_WITH_INFO. A subsequent call to **SQLError** indicates that a truncation occurred. The application can compare *pcbValue* to *cbValueMax* (specified in **SQLBindCol**) to determine which column or columns were truncated. If *pcbValue* is greater than or equal to *cbValueMax*, then truncation occurred.
- If the data value for the column is NULL, the driver stores SQL_NULL_DATA in *pcbValue*.

Tip: *The SQL_MAX_LENGTH statement option is intended to reduce network traffic and might not be supported by all drivers. To guarantee that data is truncated, an application should allocate a buffer of the desired size and specify this size in the cbValueMax argument.*

SQLFetch is valid only after a call that returns a result set.

For information about conversions allowed by **SQLBindCol** and **SQLGetData**, see "Converting Data from SQL to C" on page B-19.

Code Example

See **SQLBindCol**, **SQLColumns**, **SQLProcedures**, and **SQLGetData**.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning information about a column in a result set	**SQLDescribeCol**
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Freeing a statement handle	**SQLFreeStmt**
Fetching part or all of a column of data	**SQLGetData**
Returning the number of result set columns	**SQLNumResultCols**
Preparing a statement for execution	**SQLPrepare**

♦

Level 2

SQLForeignKeys

SQLForeignKeys can return either of the following items:

- A list of foreign keys in the specified table (columns in the specified table that refer to primary keys in other tables)
- A list of foreign keys in other tables that refer to the primary key in the specified table

The driver returns each list as a result set on the specified *hstmt*.

Syntax

```
RETCODE SQLForeignKeys(hstmt, szPkTableQualifier,
cbPkTableQualifier, szPkTableOwner, cbPkTableOwner,
szPkTableName, cbPkTableName, szFkTableQualifier,
cbFkTableQualifier, szFkTableOwner, cbFkTableOwner,
szFkTableName, cbFkTableName)
```

The **SQLForeignKeys** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szPkTableQualifier*	Input	Primary key table qualifier. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbPkTableQualifier*	Input	Length of *szPkTableQualifier.*
UCHAR FAR *	*szPkTableOwner*	Input	Primary key owner name. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have owners.
SWORD	*cbPkTableOwner*	Input	Length of *szPkTableOwner.*

(1 of 2)

Type	Argument	Use	Description
UCHAR FAR *	*szPkTableName*	Input	Primary key table name.
SWORD	*cbPkTableName*	Input	Length of *szPkTableName*.
UCHAR FAR *	*szFkTableQualifier*	Input	Foreign key table qualifier. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbFkTableQualifier*	Input	Length of *szFkTableQualifier*.
UCHAR FAR *	*szFkTableOwner*	Input	Foreign key owner name. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have owners.
SWORD	*cbFkTableOwner*	Input	Length of *szFkTableOwner*.
UCHAR FAR *	*szFkTableName*	Input	Foreign key table name.
SWORD	*cbFkTableName*	Input	Length of *szFkTableName*.

(2 of 2)

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
24000	Invalid cursor state	(DM) A cursor was open on the *hstmt* and **SQLFetch** or **SQLExtendedFetch** had been called. A cursor was open on the *hstmt* but **SQLFetch** or **SQLExtendedFetch** had not been called.
IM001	Driver does not support this function	(DM) The driver associated with the *hstmt* does not support the function.
S1000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory allocation failure	The driver was unable to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value	(DM) The arguments *szPkTableName* and *szFkTa-bleName* were both null pointers.

(1 of 2)

SQLSTATE	Error	Description
S1010	Function sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value of one of the name length arguments was less than 0, but not equal to SQL_NTS. The value of one of the name length arguments exceeded the maximum length value for the corresponding qualifier or name (see "Comments").
S1C00	Driver not capable	A table qualifier was specified and the driver or data source does not support qualifiers. A table owner was specified and the driver or data source does not support owners. The drive or data source did not support the combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options.
S1T00	Timeout expired	The timeout period expired before the data source returned the result set. The timeout period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Comments

If *szPkTableName* contains a table name, **SQLForeignKeys** returns a result set containing the primary key of the specified table and all of the foreign keys that refer to it.

If *szFkTableName* contains a table name, **SQLForeignKeys** returns a result set containing all of the foreign keys in the specified table and the primary keys (in other tables) to which they refer.

If both *szPkTableName* and *szFkTableName* contain table names, **SQLForeignKeys** returns the foreign keys in the table specified in *szFkTableName* that refer to the primary key of the table specified in *szPkTableName*. These foreign keys returned by **SQLForeignKeys** should be one key at most.

SQLForeignKeys returns results as a standard result set. If the foreign keys associated with a primary key are requested, the result set is ordered by FKTABLE_QUALIFIER, FKTABLE_OWNER, FKTABLE_NAME, and KEY_SEQ. If the primary keys associated with a foreign key are requested, the result set is ordered by PKTABLE_QUALIFIER, PKTABLE_OWNER, PKTABLE_NAME, and KEY_SEQ. The following table lists the columns in the result set.

The lengths of VARCHAR columns that the table shows are maximums; the actual lengths depend on the data source. To determine the actual lengths of the TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, and COLUMN_NAME columns, an application can call **SQLGetInfo** with the SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

Column Name	Data Type	Comments
PKTABLE_QUALIFIER	Varchar(128)	Primary key table qualifier identifier; NULL if not applicable to the data source. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have qualifiers.
PKTABLE_OWNER	Varchar(128)	Primary key table owner identifier; NULL if not applicable to the data source. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have owners.
PKTABLE_NAME	Varchar(128) not NULL	Primary key table identifier.
PKCOLUMN_NAME	Varchar(128) not NULL	Primary key column identifier.
FKTABLE_QUALIFIER	Varchar(128)	Foreign key table qualifier identifier; NULL if not applicable to the data source. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have qualifiers.

(1 of 2)

Column Name	Data Type	Comments
FKTABLE_OWNER	Varchar(128)	Foreign key table owner identifier; NULL if not applicable to the data source. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have owners.
FKTABLE_NAME	Varchar(128) not NULL	Foreign key table identifier.
FKCOLUMN_NAME	Varchar(128) not NULL	Foreign key column identifier.
KEY_SEQ	Smallint not NULL	Column sequence number in key (starting with 1).
UPDATE_RULE	Smallint	Action to be applied to the foreign key when the SQL operation is UPDATE: SQL_CASCADE SQL_RESTRICT SQL_SET_NULL NULL if not applicable to the data source.
DELETE_RULE	Smallint	Action to be applied to the foreign key when the SQL operation is DELETE: SQL_CASCADE SQL_RESTRICT SQL_SET_NULL NULL if not applicable to the data source.
FK_NAME	Varchar(128)	Foreign key identifier. NULL if not applicable to the data source.
PK_NAME	Varchar(128)	Primary key identifier. NULL if not applicable to the data source.

(2 of 2)

Code Example

The code example in this section uses four database tables. The following table lists the database tables and the columns in each database table.

Database Table	Columns
SALES_ORDER	SALES_ID CUSTOMER_ID EMPLOYEE_ID TOTAL_PRICE
SALES_LINE	SALES_ID LINE_NUMBER PART_ID QUANTITY PRICE
CUSTOMER	CUSTOMER_ID CUST_NAME ADDRESS PHONE
EMPLOYEE	EMPLOYEE_ID NAME AGE BIRTHDAY

In the SALES_ORDER table, CUSTOMER_ID identifies the customer to whom the sale was made. It is a foreign key that refers to CUSTOMER_ID in the CUSTOMER table. In the SALES_ORDER table, EMPLOYEE_ID identifies the employee who made the sale. It is a foreign key that refers to EMPLOYEE_ID in the EMPLOYEE table.

In the SALES_LINE table, SALES_ID identifies the sales order with which the line item is associated. It is a foreign key that refers to SALES_ID in the SALES_ORDER table.

The code example calls **SQLPrimaryKeys** to get the primary key of the SALES_ORDER table. The result set has one row and the significant columns are as follows.

TABLE_NAME	COLUMN_NAME	KEY_SEQ
SALES_ORDER	SALES_ID	1

Next, the code example calls **SQLForeignKeys** to get the foreign keys in other tables that reference the primary key of the SALES_ORDER table. The result set has one row and the significant columns are as follows.

PKTABLE_NAME	PKCOLUMN_NAME	FKTABLE_NAME	FKCOLUMN_NAME	KEY_SEQ
SALES_ORDER	SALES_ID	SALES_LINE	SALES_ID	1

Finally, the code example calls **SQLForeignKeys** to get the foreign keys in the SALES_ORDER table that refer to the primary keys of other tables. The result set has two rows and the significant columns are as follows.

PKTABLE_NAME	PKCOLUMN_NAME	FKTABLE_NAME	FKCOLUMN_NAME	KEY_SEQ
CUSTOMER	CUSTOMER_ID	SALES_ORDER	CUSTOMER_ID	1
EMPLOYEE	EMPLOYEE_ID	SALES_ORDER	EMPLOYEE_ID	1

```
#define TAB_LEN SQL_MAX_TABLE_NAME_LEN + 1
#define COL_LEN SQL_MAX_COLUMN_NAME_LEN + 1

LPSTRszTable;             /* Table to display       */

UCHARszPkTable[TAB_LEN];  /* Primary key table name */
UCHARszFkTable[TAB_LEN];  /* Foreign key table name */
UCHARszPkCol[COL_LEN];    /* Primary key column     */
UCHARszFkCol[COL_LEN];    /* Foreign key column     */

HSTMT   hstmt;
SDWORD  cbPkTable, cbPkCol, cbFkTable, cbFkCol, cbKeySeq;
SWORD   iKeySeq;
RETCODE retcode;

/* Bind the columns that describe the primary and foreign keys.  */
/* Ignore the table owner, name, and qualifier for this example. */

SQLBindCol(hstmt, 3, SQL_C_CHAR, szPkTable, TAB_LEN, &cbPkTable);
SQLBindCol(hstmt, 4, SQL_C_CHAR, szPkCol, COL_LEN, &cbPkCol);
SQLBindCol(hstmt, 5, SQL_C_SSHORT, &iKeySeq, TAB_LEN, &cbKeySeq);
SQLBindCol(hstmt, 7, SQL_C_CHAR, szFkTable, TAB_LEN, &cbFkTable);
SQLBindCol(hstmt, 8, SQL_C_CHAR, szFkCol, COL_LEN, &cbFkCol);
strcpy(szTable, "SALES_ORDER");
/* Get the names of the columns in the primary key.              */

retcode = SQLPrimaryKeys(hstmt,
                         NULL, 0,           /* Table qualifier   */
                         NULL, 0,           /* Table owner       */
                         szTable, SQL_NTS); /* Table name        */

while ((retcode == SQL_SUCCESS) || (retcode == SQL SUCCESS_WITH_INFO)) {

    /* Fetch and display the result set. This will be a list of the */
    /* columns in the primary key of the SALES_ORDER table.         */

    retcode = SQLFetch(hstmt);
    if (retcode == SQL_SUCCESS || retcode != SQL_SUCCESS_WITH_INFO)
        fprintf(out, "Column: %s    Key Seq: %hd \n", szPkCol, iKeySeq);
}
/* Close the cursor (the hstmt is still allocated).                */

SQLFreeStmt(hstmt, SQL_CLOSE);

/* Get all the foreign keys that refer to SALES_ORDER primary key. */

retcode = SQLForeignKeys(hstmt,
                         NULL, 0,          /* Primary qualifier    */
                         NULL, 0,          /* Primary owner        */
                         szTable, SQL_NTS, /* Primary table        */
                         NULL, 0,          /* Foreign qualifier    */
                         NULL, 0,          /* Foreign owner        */
                         NULL, 0);         /* Foreign table        */

while ((retcode == SQL_SUCCESS) || (retcode == SQL_SUCCESS_WITH_INFO)) {
    /* Fetch and display the result set. This will be all of the    */
    /* foreign keys in other tables that refer to the SALES_ORDER   */
    /* primary key.                                                 */

    retcode = SQLFetch(hstmt);
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
        fprintf(out, "%-s ( %-s ) <-- %-s ( %-s )\n", szPkTable,

                szPkCol, szFkTable, szFkCol);
}
/* Close the cursor (the hstmt is still allocated).                 */

SQLFreeStmt(hstmt, SQL_CLOSE);

/* Get all the foreign keys in the SALES_ORDER table.               */

retcode = SQLForeignKeys(hstmt,
                         NULL, 0,           /* Primary qualifier    */
                         NULL, 0,           /* Primary owner        */
                         NULL, 0,           /* Primary table        */
                         NULL, 0,           /* Foreign qualifier    */
                         NULL, 0,           /* Foreign owner        */
                         szTable, SQL_NTS); /* Foreign table        */

while ((retcode == SQL_SUCCESS) || (retcode == SQL_SUCCESS_WITH_INFO)) {

    /* Fetch and display the result set. This will be all of the    */
    /* primary keys in other tables that are referred to by foreign */
    /* keys in the SALES_ORDER table.                               */

    retcode = SQLFetch(hstmt);
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
        fprintf(out, "%-s ( %-s )--> %-s ( %-s )\n", szFkTable, szFkCol,
                szPkTable, szPkCol);
}

/* Free the hstmt. */

SQLFreeStmt(hstmt, SQL_DROP);
```

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch** (extension)
Fetching a row of data	**SQLFetch**
Returning the columns of a primary key	**SQLPrimaryKeys** (extension)
Returning table statistics and indexes	**SQLStatistics** (extension)

♦

Core

SQLFreeConnect

SQLFreeConnect releases a connection handle and frees all memory associated with the handle.

Syntax

```
RETCODE SQLFreeConnect(hdbc)
```

The **SQLFreeConnect** function accepts the following argument.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1010	Function-sequence error	(DM) The function was called prior to calling **SQLDisconnect** for the *hdbc*.

Usage

Prior to calling **SQLFreeConnect,** an application must call **SQLDisconnect** for the *hdbc*. Otherwise, **SQLFreeConnect** returns SQL_ERROR, and the *hdbc* remains valid. **SQLDisconnect** automatically drops any *hstmts* open on the *hdbc*.

Code Example

See **SQLBrowseConnect** and **SQLConnect**.

Related Functions

For Information About	See
Allocating a statement handle	**SQLAllocConnect**
Connecting to a data source	**SQLConnect**
Disconnecting from a data source	**SQLDisconnect**
Connecting to a data source using a connection string or dialog box	**SQLDriverConnect**
Freeing an environment handle	**SQLFreeEnv**
Freeing a statement handle	**SQLFreeStmt**

♦

Core

SQLFreeEnv

SQLFreeEnv frees the environment handle and releases all memory associated with the environment handle.

Syntax

```
RETCODE SQLFreeEnv(henv)
```

The **SQLFreeEnv** function accepts the following argument.

Type	Argument	Use	Description
HENV	*henv*	Input	Environment handle

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1010	Function-sequence error	(DM) There was at least one *hdbc* in an allocated or connected state. Call **SQLDisconnect** and **SQLFreeConnect** for each *hdbc* before calling **SQLFreeEnv**.

Usage

Before an application calls **SQLFreeEnv**, it must call **SQLFreeConnect** for any *hdbc* allocated under the *henv*. Otherwise, **SQLFreeEnv** returns SQL_ERROR and the *henv* and any active *hdbc* remains valid.

When INFORMIX-CLI processes the **SQLFreeEnv** function, it checks the TraceAutoStop value in the initialization files. If the value is 1, the INFORMIX-CLI disables tracing for all applications and sets the TraceAutoStop value in the initialization files to 0.

Code Example

See **SQLBrowseConnect** and **SQLConnect**.

Related Functions

For Information About	See
Allocating an environment handle	**SQLAllocEnv**
Freeing a connection handle	**SQLFreeConnect**

♦

Core

SQLFreeStmt

SQLFreeStmt stops the processing that is associated with a specific *hstmt*, closes any open cursors that are associated with the *hstmt*, discards pending results, and, optionally, frees all resources associated with the statement handle.

Syntax

```
RETCODE SQLFreeStmt(hstmt, fOption)
```

The **SQLFreeStmt** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UWORD	*fOption*	Input	One of the following options: ■ SQL_CLOSE: Close the cursor associated with *hstmt* (if one is defined) and discard all pending results. The application can reopen this cursor later by executing a SELECT statement again with the same or different parameter values. If no cursor is open, this option has no effect for the application. ■ SQL_DROP: Release the *hstmt*, free all resources associated with it, close the cursor (if one is open), and discard all pending rows. This option terminates all access to the *hstmt*. The *hstmt* must be reallocated to be reused. ■ SQL_UNBIND: Release all column buffers bound by **SQLBindCol** for the given *hstmt*. ■ SQL_RESET_PARAMS: Release all parameter buffers set by **SQLBindParameter** for the given *hstmt*.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1092	Option type out of range	(DM) The value specified for the argument *fOption* was not: SQL_CLOSE SQL_DROP SQL_UNBIND SQL_RESET_PARAMS

Usage

An application can call **SQLFreeStmt** to terminate processing of a SELECT statement with or without canceling the statement handle.

The SQL_DROP option frees all resources that the **SQLAllocStmt** function allocates.

Code Example

See **SQLBrowseConnect** and **SQLConnect**.

Related Functions

For Information About	See
Allocating a statement handle	**SQLAllocStmt**
Canceling statement processing	**SQLCancel**
Setting a cursor name	**SQLSetCursorName**

♦

Level 1

SQLGetConnectOption

SQLGetConnectOption returns the current setting of a connection option.

Syntax

```
RETCODE SQLGetConnectOption(hdbc, fOption, pvParam)
```

The **SQLGetConnectOption** function accepts the following arguments.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.
UWORD	*fOption*	Input	Option to retrieve.
PTR	*pvParam*	Output	Value associated with *fOption*. Depending on the value of *fOption*, a 32-bit integer value or a pointer to a null-terminated character string will be returned in *pvParam*.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	(DM) An *fOption* value was specified that required an open connection.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) **SQLBrowseConnect** was called for the *hdbc* and returned SQL_NEED_DATA. This function was called before **SQLBrowseConnect** returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.
S1092	Option type out of range	(DM) The value specified for the argument *fOption* was in the block of numbers reserved for ODBC connection and statement options but was not valid for the version of ODBC supported by the driver.
S1C00	Driver not capable	The driver or data source does not support the value specified for the argument *f*Option.

Usage

For a list of options, see **SQLSetConnectOption**.

Important: *When fOption specifies an option that returns a string, pvParam must be a pointer to storage for the string. The maximum length of the string is SQL_MAX_OPTION_STRING_LENGTH bytes (excluding the null-termination byte).*

Depending on the option, an application does not need to establish a connection prior to calling **SQLGetConnectOption**. However, if **SQLGetConnectOption** is called and the specified option does not have a default and has not been set by a prior call to **SQLSetConnectOption**, **SQLGetConnectOption** returns SQL_NO_DATA_FOUND.

Although an application can set statement options using **SQLSetConnectOption**, an application cannot use **SQLGetConnectOption** to retrieve statement-option values; it must call **SQLGetStmtOption** to retrieve the settings of statement options.

Related Functions

For Information About	See
Returning the setting of a statement option	**SQLGetStmtOption**
Setting a connection option	**SQLSetConnectOption**
Setting a statement option	**SQLSetStmtOption**

♦

Core

SQLGetCursorName

SQLGetCursorName returns the cursor name associated with a specified *hstmt*.

Syntax

```
RETCODE SQLGetCursorName(hstmt, szCursor, cbCursorMax,
pcbCursor)
```

The **SQLGetCursorName** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szCursor*	Output	Pointer to storage for the cursor name.
SWORD	*cbCursorMax*	Input	Length of *szCursor*.
SWORD FAR *	*pcbCursor*	Output	Total number of bytes (excluding the null termination byte) available to return in *szCursor*. If the number of bytes available to return is greater than or equal to *cbCursorMax*, the cursor name in *szCursor* is truncated to *cbCursorMax* – 1 bytes.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The buffer *szCursor* was not large enough to return the entire cursor name, so the cursor name was truncated. The argument *pcbCursor* contains the length of the untruncated cursor name (function returns SQL_SUCCESS_WITH_INFO).
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate sufficient memory to execute or complete the function.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1015	No cursor name available	(DM) There was no open cursor on the *hstmt*, and no cursor name had been set with **SQLSetCursorName**.
S1090	Invalid string or buffer length	(DM) The value specified in the argument *cbCursorMax* was less than 0.

Usage

The only INFORMIX-CLI SQL statements that use a cursor name are positioned update and delete (for example, UPDATE *table-name*...WHERE CURRENT OF *cursor-name*). If the application does not call **SQLSetCursorName** to define a cursor name when a SELECT statement executes, the driver generates a name that begins with the letters SQL_CUR and does not exceed 18 characters.

SQLGetCursorName returns the name of a cursor regardless of whether the name was created explicitly or implicitly.

A cursor name that is set either explicitly or implicitly remains set until the *hstmt* with which it is associated is dropped, using **SQLFreeStmt** with the SQL_DROP option.

Related Functions

For Information About	See
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Preparing a statement for execution	**SQLPrepare**
Setting a cursor name	**SQLSetCursorName**
Setting cursor scrolling options	**SQLSetScrollOptions**

♦

Level 1

SQLGetData

SQLGetData returns result data for a single unbound column in the current row. The application must call **SQLFetch** or **SQLExtendedFetch** to position the cursor on a row of data before it calls **SQLGetData**. It is possible to use **SQLBindCol** for some columns and use **SQLGetData** for others within the same row. This function can be used to retrieve character or binary data values in parts from a column with a character, binary, or data-source-specific data type (for example, data from SQL_LONGVARBINARY or SQL_LONGVARCHAR columns).

Syntax

```
RETCODE SQLGetData(hstmt, icol, fCType, rgbValue, cbValueMax,
pcbValue)
```

The **SQLGetData** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UWORD	*icol*	Input	Column number of result data, ordered sequentially left to right, starting at 1.
SWORD	*fCType*	Input	The result data's INFORMIX-CLI C data type. The possible values are any of the *fCType* values listed in Appendix B, "Data Types." as well as SQL_C_DEFAULT. For more information about data types and conversions, see Appendix B.
PTR	*rgbValue*	Output	Pointer to storage for the data.
SDWORD	*cbValueMax*	Input	Maximum length of the *rgbValue* buffer. For character data, *rgbValue* must also include space for the null-termination byte. For character and binary C data, *cbValueMax* determines the amount of data that can be received in a single call to **SQLGetData**. For all other types of C data, *cbValueMax* is ignored; the driver assumes that the size of *rgbValue* is the size of the C data type specified with *fCType* and returns the entire data value. For more information about length, see "Precision, Scale, Length, and Display Size" on page B-5.

(1 of 2)

Type	Argument	Use	Description
SDWORD FAR *	*pcbValue*	Output	SQL_NULL_DATA, the total number of bytes (excluding the null-termination byte for character data) available to return in *rgbValue* prior to the current call to **SQLGetData**, or SQL_NO_TOTAL if the number of available bytes cannot be determined. For character data, if *pcbValue* is SQL_NO_TOTAL or is greater than or equal to *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax* – 1 bytes and is null-terminated by the driver. For binary data, if *pcbValue* is SQL_NO_TOTAL or is greater than *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax* bytes. For all other data types, the value of *cbValueMax* is ignored, and the driver assumes the size of *rgbValue* is the size of the C data type specified with *fCType*.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	All the data for the specified column, *icol*, could not be retrieved in a single call to the function. The argument *pcbValue* contains the length of the data that remains in the specified column prior to the current call to **SQLGetData** (function returns SQL_SUCCESS_WITH_INFO). For more information on using multiple calls to **SQLGetData** for a single column, see "Usage" on page 12-174.
07006	Restricted data-type attribute violation	The data value cannot be converted to the C data type specified by the argument *fCType*.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
22002	Indicator value required but not supplied	If the column value is null, then *StrLen_or_Ind* must not be a null pointer.
22003	Numeric value out of range	Returning the numeric value (as numeric or string) for the column would cause the whole (as opposed to fractional) number to truncate. Returning the binary value for the column would have caused a loss of binary significance.
22005	Error in assignment	The data for the column was incompatible with the data type in which it was to be converted. For more information, see Appendix B, "Data Types."
22008	Datetime-field overflow	The data for the column was not a valid date, time, or time-stamp value. For more information, see Appendix B.
22012	Division by zero	A value from an arithmetic expression was returned that resulted in a division by zero.

(1 of 3)

SQLSTATE	Error	Description
24000	Invalid cursor state	(DM) The *hstmt* was in an executed state, but no result set was associated with the *hstmt*. (DM) A cursor was open on the *hstmt,* but **SQLFetch** or **SQLExtendedFetch** was not called. A cursor was open on the *hstmt* and **SQLFetch** or **SQLExtendedFetch** had been called, but the cursor was positioned before the start of the result set or after the end of the result set.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1002	Invalid column number	The value specified for the argument *icol* was 0, and **SQLFetch** was used to fetch the data. The value specified for the argument *icol* was 0, and the SQL_USE_BOOKMARKS statement option was set to SQL_UB_OFF. The specified column was greater than the number of result columns. The specified column was bound through a call to **SQLBindCol**. This description does not apply to drivers that return the SQL_GD_BOUND bitmask for the SQL_GETDATA_EXTENSIONS option in **SQLGetInfo**. The specified column was at or before the last bound column specified through **SQLBindCol**. This description does not apply to drivers that return the SQL_GD_ANY_COLUMN bitmask for the SQL_GETDATA_EXTENSIONS option in **SQLGetInfo**. The application already called **SQLGetData** for the current row. The column specified in the current call was before the column specified in the preceding call. This description does not apply to drivers that return the SQL_GD_ANY_ORDER bitmask for the SQL_GETDATA_EXTENSIONS option in **SQLGetInfo**.
S1003	Program type out of range	(DM) The argument *fCType* was not a valid data type or SQL_C_DEFAULT.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.

(2 of 3)

SQLSTATE	Error	Description
S1009	Invalid argument value	(DM) The argument *rgbValue* was a null pointer.
S1010	Function-sequence error	(DM) The specified *hstmt* was not in an executed state. The function was called without first calling **SQLExecDirect**, **SQLExecute**, or a catalog function. (DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value specified for argument *cbValueMax* was less than 0.
S1109	Invalid cursor position	The cursor was positioned (by **SQLExtendedFetch**) on a row for which the value in the *rgfRowStatus* array in **SQLExtendedFetch** was SQL_ROW_DELETED or SQL_ROW_ERROR.
S1C00	Driver not capable	The driver or data source does not support the use of **SQLGetData** with multiple rows in **SQLExtendedFetch**. This description does not apply to drivers that return the SQL_GD_BLOCK bitmask for the SQL_GETDATA_EXTENSIONS option in **SQLGetInfo**. The driver or data source does not support the conversion specified by the combination of the *fCType* argument and the SQL data type of the corresponding column. This error applies only when the SQL data type of the column was mapped to a driver-specific SQL data type. The argument *icol* was 0, and the driver does not support bookmarks.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(3 of 3)

Usage

With each call, the driver sets *pcbValue* to the number of bytes that are available in the result column before the current call to **SQLGetData**. If **SQLSetStmtOption** sets SQL_MAX_LENGTH and the total number of bytes that are available on the first call is greater than SQL_MAX_LENGTH, the available number of bytes is set to SQL_MAX_LENGTH.

The SQL_MAX_LENGTH statement option is intended to reduce network traffic and might not be supported by all drivers. To guarantee that data is truncated, an application should allocate a buffer of the desired size and specify this size in the *cbValueMax* argument. If the total number of bytes that are in the result column cannot be determined in advance, the driver sets *pcbValue* to SQL_NO_TOTAL. If the data value for the column is null, the driver stores SQL_NULL_DATA in *pcbValue*.

SQLGetData can convert data to a different data type. The result and success of the conversion is determined by the rules for assignment specified in "Converting Data from SQL to C Data Types" in Appendix D, "Data Types."

If the application requires more than one call to **SQLGetData** to retrieve data from a single column with a character, binary, or data source-specific data type, the driver returns SQL_SUCCESS_WITH_INFO. A subsequent call to **SQLError** returns SQLSTATE 01004 (Data truncated). The application can then use the same column number to retrieve subsequent parts of the data until **SQLGetData** returns SQL_SUCCESS, indicating that all the column data was retrieved. **SQLGetData** returns SQL_NO_DATA_FOUND when it calls for a column after all the data is retrieved and before data is retrieved for a subsequent column. The application can ignore excess data by proceeding to the next result column.

Important: *An application can use **SQLGetData** to retrieve data from a column in parts only when it retrieves character C data from a column with a character, binary, or data-source-specific data type or when it retrieves binary C data from a column with a character, binary, or data-source-specific data type. If **SQLGetData** is called more than once in a row for a column under any other conditions, it returns SQL_NO_DATA_FOUND for all calls after the first call.*

For maximum interoperability, applications should call **SQLGetData** only for unbound columns with numbers greater than the number of the last bound column. Within a single row of data, the column number in each call to **SQLGetData** should be greater than or equal to the column number in the previous call (that is, data should be retrieved in increasing order of column number). As extended functionality, drivers can return data through **SQLGetData** from bound columns, from columns before the last bound column, or from columns in any order. To determine whether a driver supports these extensions, an application calls **SQLGetInfo** with the SQL_GETDATA_EXTENSIONS option.

Furthermore, applications that use **SQLExtendedFetch** to retrieve data can call **SQLGetData** only when the row set size is 1.

Code Example

In the following example, an application executes a SELECT statement to return a result set of the employee names, ages, and birthdays sorted by birthday, age, and name. For each row of data, it calls **SQLFetch** to position the cursor to the next row. It calls **SQLGetData** to retrieve the fetched data; the storage locations for the data and the returned number of bytes are specified in the call to **SQLGetData**. Finally, it prints each employee's name, age, and birthday.

```
#define NAME_LEN 30
#define BDAY_LEN 11

UCHAR     szName[NAME_LEN], szBirthday[BDAY_LEN];
SWORD     sAge;
SDWORD    cbName, cbAge, cbBirthday;

retcode = SQLExecDirect(hstmt,

            "SELECT NAME, AGE, BIRTHDAY FROM EMPLOYEE ORDER BY 3, 2, 1",
            SQL_NTS);

if (retcode = = SQL_SUCCESS) {
    while (TRUE) {
        retcode = SQLFetch(hstmt);
        if (retcode = = SQL_ERROR || retcode = = SQL_SUCCESS_WITH_INFO) {
            show_error();
        }
        if (retcode = = SQL_SUCCESS || retcode = = SQL_SUCCESS_WITH_INFO){
            /* Get data for columns 1, 2, and 3 */
            /* Print the row of data            */

            SQLGetData(hstmt, 1, SQL_C_CHAR, szName, NAME_LEN, &cbName);
            SQLGetData(hstmt, 2, SQL_C_SSHORT, &sAge, 0, &cbAge);
            SQLGetData(hstmt, 3, SQL_C_CHAR, szBirthday, BDAY_LEN,

                        &cbBirthday);

            fprintf(out, "%-*s %-2d %*s", NAME_LEN-1, szName, sAge,

                     BDAY_LEN-1, szBirthday);
        } else {
            break;
        }
    }
}
```

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Sending parameter data at execution time	**SQLPutData**

♦

Level 1

SQLGetFunctions

SQLGetFunctions returns information about whether the INFORMIX-CLI driver supports a specific function.

Syntax

```
RETCODE SQLGetFunctions(hdbc, fFunction, pfExists)
```

The **SQLGetFunctions** function accepts the following arguments.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.
UWORD	*fFunction*	Input	SQL_API_ALL_FUNCTIONS or a **#define** value that identifies the ODBC function of interest. For a list of **#define** values that identify ODBC functions, see "Usage" on page 12-180.
UWORD FAR *	*pfExists*	Output	If *fFunction* is SQL_API_ALL_FUNCTIONS, *pfExists* points to a UWORD array with 100 elements. The array is indexed by **#define** values used by *fFunction* to identify each ODBC function; some elements of the array are unused and are reserved for future use. An element is TRUE if it identifies an ODBC function supported by the driver. It is FALSE if it identifies an ODBC function not supported by the driver or does not identify an ODBC function. The *fFunction* value SQL_API_ALL_FUNCTIONS was added in ODBC 2.0. If *fFunction* identifies a single ODBC function, *pfExists* points to a single UWORD. *pfExists* is TRUE if the specified function is supported by the driver; otherwise, it is FALSE.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) **SQLGetFunctions** was called before **SQLConnect**, **SQLBrowseConnect**, or **SQLDriverConnect**. (DM) **SQLBrowseConnect** was called for the *hdbc* and returned SQL_NEED_DATA. This function was called before **SQLBrowseConnect** returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.
S1095	Function type out of range	(DM) An invalid *fFunction* value was specified.

Usage

The following list identifies valid values for *fFunction* for ODBC core functions that the INFORMIX-CLI driver supports:

- SQL_API_SQLALLOCCONNECT
- SQL_API_SQLALLOCENV
- SQL_API_SQLALLOCSTMT
- SQL_API_SQLBINDCOL
- SQL_API_SQLCANCEL
- SQL_API_SQLCOLATTRIBUTES
- SQL_API_SQLCONNECT
- SQL_API_SQLDESCRIBECOL
- SQL_API_SQLDISCONNECT
- SQL_API_SQLERROR
- SQL_API_SQLEXECDIRECT
- SQL_API_SQLEXECUTE
- SQL_API_SQLFETCH
- SQL_API_SQLFREECONNECT
- SQL_API_SQLFREEENV
- SQL_API_SQLFREESTMT
- SQL_API_SQLGETCURSORNAME
- SQL_API_SQLNUMRESULTCOLS
- SQL_API_SQLPREPARE
- SQL_API_SQLROWCOUNT
- SQL_API_SQLSETCURSORNAME
- SQL_API_SQLSETPARAM
- SQL_API_SQLTRANSACT

The following list identifies valid values for *fFunction* for ODBC extension Level 1 functions that the INFORMIX-CLI driver supports:

- SQL_API_SQLBINDPARAMETER
- SQL_API_SQLCOLUMNS
- SQL_API_SQLDRIVERCONNECT

- SQL_API_SQLGETCONNECTOPTION
- SQL_API_SQLGETDATA
- SQL_API_SQLGETFUNCTIONS
- SQL_API_SQLGETINFO
- SQL_API_SQLGETSTMTOPTION
- SQL_API_SQLGETTYPEINFO
- SQL_API_SQLPARAMDATA
- SQL_API_SQLPUTDATA
- SQL_API_SQLSETCONNECTOPTION
- SQL_API_SQLSETSTMTOPTION
- SQL_API_SQLSPECIALCOLUMNS
- SQL_API_SQLSTATISTICS
- SQL_API_SQLTABLES

The following list identifies valid values for *fFunction* for ODBC extension Level 2 functions that the INFORMIX-CLI driver supports:

- SQL_API_SQLBROWSECONNECT
- SQL_API_SQLCOLUMNPRIVILEGES
- SQL_API_SQLDATASOURCES
- SQL_API_SQLDESCRIBEPARAM
- SQL_API_SQLDRIVERS
- SQL_API_SQLEXTENDEDFETCH
- SQL_API_SQLFOREIGNKEYS
- SQL_API_SQLMORERESULTS
- SQL_API_SQLNATIVESQL
- SQL_API_SQLNUMPARAMS
- SQL_API_SQLPRIMARYKEYS
- SQL_API_SQLPROCEDURECOLUMNS
- SQL_API_SQLPROCEDURES
- SQL_API_SQLSETSCROLLOPTIONS
- SQL_API_SQLTABLEPRIVILEGES

Code Example

The following examples show how an application uses **SQLGetFunctions** to determine whether a driver supports **SQLTables**, **SQLColumns**, and **SQLStatistics**. If the driver does not support these functions, the application disconnects from the driver. The first example calls **SQLGetFunctions** once for each function.

```
UWORD TablesExists, ColumnsExists, StatisticsExists;

SQLGetFunctions(hdbc, SQL_API_SQLTABLES, &TablesExists);
SQLGetFunctions(hdbc, SQL_API_SQLCOLUMNS, &ColumnsExists);
SQLGetFunctions(hdbc, SQL_API_SQLSTATISTICS,
&StatisticsExists);

if (TablesExists && ColumnsExists && StatisticsExists) {

    /* Continue with application */

}

SQLDisconnect(hdbc);
```

The second example calls **SQLGetFunctions** once and passes it an array in which **SQLGetFunctions** returns information about all INFORMIX-CLI functions.

```
UWORD fExists[100];

SQLGetFunctions(hdbc, SQL_API_ALL_FUNCTIONS, fExists);

if (fExists[SQL_API_SQLTABLES] &&
    fExists[SQL_API_SQLCOLUMNS] &&
    fExists[SQL_API_SQLSTATISTICS]) {

    /* Continue with application */

}

SQLDisconnect(hdbc);
```

Related Functions

For Information About	See
Returning the setting of a connection option	**SQLGetConnectOption**
Returning information about a driver or data source	**SQLGetInfo**
Returning the setting of a statement option	**SQLGetStmtOption**

♦

Level 1

SQLGetInfo

SQLGetInfo returns general information about the driver and data source associated with an *hdbc*.

Syntax

```
RETCODE SQLGetInfo(hdbc, fInfoType, rgbInfoValue,
cbInfoValueMax, pcbInfoValue)
```

The **SQLGetInfo** function accepts the following arguments.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.
UWORD	*fInfoType*	Input	Type of information. The *fInfoType* argument must be a value that represents the type of interest. For more information, see "Usage" on page 12-187.

(1 of 2)

Type	Argument	Use	Description
PTR	*rgbInfoValue*	Output	Pointer to storage for the information. Depending on the *fInfoType* requested, the information returned will be one of the following: a null-terminated character string, a 16-bit integer value, a 32-bit flag, or a 32-bit binary value.
SWORD	*cbInfoValueMax*	Input	Maximum length of the *rgbInfoValue* buffer.
SWORD FAR *	*pcbInfoValue*	Output	The total number of bytes (excluding the null-termination byte for character data) available to return in *rgbInfoValue*. For character data, if the number of bytes available to return is greater than or equal to *cbInfoValueMax*, the information in *rgbInfoValue* is truncated to *cbInfoValueMax* – 1 bytes and is null-terminated by the driver. For all other types of data, the value of *cbValueMax* is ignored, and the driver assumes the size of *rgbValue* is 32 bits.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The buffer *rgbInfoValue* was not large enough to return all of the requested information, so the information was truncated. The argument *pcbInfoValue* contains the length of the requested information in its untruncated form (function returns SQL_SUCCESS_WITH_INFO).
08003	Connection not open	(DM) The type of information requested in *fInfoType* requires an open connection. Of the information types reserved by ODBC, only SQL_ODBC_VER can be returned without an open connection.
22003	Numeric value out of range	Returning the requested information would cause a loss of numeric or binary significance.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1009	Invalid argument value	(DM) The *fInfoType* was SQL_DRIVER_HSTMT, and the value pointed to by *rgbInfoValue* was not a valid statement handle.

(1 of 2)

SQLSTATE	Error	Description
S1090	Invalid string or buffer length	(DM) The value specified for argument *cbInfoValueMax* was less than 0.
S1096	Information type out of range	(DM) The value specified for the argument *fOption* was in the block of numbers reserved for ODBC information types but was not valid for the version of ODBC that the driver supports.
S1C00	Driver not capable	The driver does not support the value specified for the argument *fOption*.
S1T00	Time-out expired	The time-out period expired before the data source returned the requested information. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

The format of the information returned in *rgbInfoValue* depends on the *fInfoType* requested. **SQLGetInfo** returns information in one of the following five formats:

- A null-terminated character string
- A 16-bit integer value
- A 32-bit bitmask
- A 32-bit integer value
- A 32-bit binary value

The format for each of the following information types is noted in the description. The application must cast the value returned in *rgbInfoValue* accordingly. For an example of how an application could retrieve data from a 32-bit bitmask, see "Code Example" on page 12-216.

A driver must return a value for each information type defined in the following tables. If an information type does not apply to the driver or data source, the driver returns one of the following values.

Format of rgbInfoValue	**Returned Value**
Character string ("Y" or "N")	`N`
Character string (not "Y" or "N")	`Empty string`
16-bit integer	0
32-bit bitmask or 32-bit binary value	`0L`

For example, if a data source does not support procedures, **SQLGetInfo** returns the following values for the *fInfoType* values, which are related to procedures.

fInfoType	**Returned Value**
SQL_PROCEDURES	`N`
SQL_ACCESSIBLE_PROCEDURES	`N`
SQL_MAX_PROCEDURE_NAME_LEN	`0`
SQL_PROCEDURE_TERM	`Empty string`

SQLGetInfo returns SQLSTATE S1096 (Invalid argument value) for a value of *fInfoType* that the INFORMIX-CLI driver does not define.

Information Types

This section lists the information types supported by **SQLGetInfo**. Information types are grouped categorically and listed alphabetically.

Driver Information

The following values of *fInfoType* return information about the INFORMIX-CLI driver, such as the number of active statements, the data source name, and the API conformance levels:

- SQL_ACTIVE_CONNECTIONS
- SQL_ACTIVE_STATEMENTS
- SQL_DATA_SOURCE_NAME
- SQL_DRIVER_HDBC
- SQL_DRIVER_HENV
- SQL_DRIVER_HLIB
- SQL_DRIVER_HSTMT
- SQL_DRIVER_NAME
- SQL_DRIVER_ODBC_VER
- SQL_DRIVER_VER
- SQL_FETCH_DIRECTION
- SQL_FILE_USAGE
- SQL_GETDATA_EXTENSIONS
- SQL_LOCK_TYPES
- SQL_ODBC_API_CONFORMANCE
- SQL_ODBC_SAG_CLI_CONFORMANCE
- SQL_ODBC_VER
- SQL_POS_OPERATIONS
- SQL_ROW_UPDATES
- SQL_SEARCH_PATTERN_ESCAPE
- SQL_SERVER_NAME

DBMS Product Information

The following values of *fInfoType* return information about the Informix DBMS product, such as the DBMS name and version:

- SQL_DATABASE_NAME
- SQL_DBMS_NAME
- SQL_DBMS_VER

Data Source Information

The following values of *fInfoType* return information about the data source, such as cursor characteristics and transaction capabilities:

- SQL_ACCESSIBLE_PROCEDURES
- SQL_ACCESSIBLE_TABLES
- SQL_CONCAT_NULL_BEHAVIOR
- SQL_CURSOR_COMMIT_BEHAVIOR
- SQL_CURSOR_ROLLBACK_BEHAVIOR
- SQL_DATA_SOURCE_READ_ONLY
- SQL_DEFAULT_TXN_ISOLATION
- SQL_MULT_RESULT_SETS
- SQL_MULTIPLE_ACTIVE_TXN
- SQL_NEED_LONG_DATA_LEN
- SQL_NULL_COLLATION
- SQL_OWNER_TERM
- SQL_PROCEDURE_TERM
- SQL_QUALIFIER_TERM
- SQL_SCROLL_CONCURRENCY
- SQL_SCROLL_OPTIONS
- SQL_STATIC_SENSITIVITY
- SQL_TABLE_TERM
- SQL_TXN_CAPABLE
- SQL_TXN_ISOLATION_OPTION
- SQL_USER_NAME

Supported SQL

The following values of *fInfoType* return information about the SQL statements that the data source supports. These information types do not exhaustively describe ODBC SQL grammar, but they describe those parts of the grammar for which data sources commonly offer different levels of support.

Applications should determine the general level of supported grammar from the SQL_ODBC_SQL_CONFORMANCE information type and use the other information types to determine variations from the stated conformance level:

- SQL_ALTER_TABLE
- SQL_COLUMN_ALIAS
- SQL_CORRELATION_NAME
- SQL_EXPRESSIONS_IN_ORDERBY
- SQL_GROUP_BY
- SQL_IDENTIFIER_CASE
- SQL_IDENTIFIER_QUOTE_CHAR
- SQL_KEYWORDS
- SQL_LIKE_ESCAPE_CLAUSE
- SQL_NON_NULLABLE_COLUMNS
- SQL_ODBC_SQL_CONFORMANCE
- SQL_ODBC_SQL_OPT_IEF
- SQL_ORDER_BY_COLUMNS_IN_SELECT
- SQL_OUTER_JOINS
- SQL_OWNER_USAGE
- SQL_POSITIONED_STATEMENTS
- SQL_PROCEDURES
- SQL_QUALIFIER_LOCATION
- SQL_QUALIFIER_NAME_SEPARATOR
- SQL_QUALIFIER_USAGE
- SQL_QUOTED_IDENTIFIER_CASE
- SQL_SPECIAL_CHARACTERS
- SQL_SUBQUERIES
- SQL_UNION

SQL Limits

The following values of *fInfoType* return information about the limits applied to identifiers and clauses in SQL statements, such as the maximum lengths of identifiers and the maximum number of columns in a SELECT list. The driver or the data source might impose limitations:

- SQL_MAX_BINARY_LITERAL_LEN
- SQL_MAX_CHAR_LITERAL_LEN
- SQL_MAX_COLUMN_NAME_LEN
- SQL_MAX_COLUMNS_IN_GROUP_BY
- SQL_MAX_COLUMNS_IN_ORDER_BY
- SQL_MAX_COLUMNS_IN_INDEX
- SQL_MAX_COLUMNS_IN_SELECT
- SQL_MAX_COLUMNS_IN_TABLE
- SQL_MAX_CURSOR_NAME_LEN
- SQL_MAX_INDEX_SIZE
- SQL_MAX_OWNER_NAME_LEN
- SQL_MAX_PROCEDURE_NAME_LEN
- SQL_MAX_QUALIFIER_NAME_LEN
- SQL_MAX_ROW_SIZE
- SQL_MAX_ROW_SIZE_INCLUDES_LONG
- SQL_MAX_STATEMENT_LEN
- SQL_MAX_TABLE_NAME_LEN
- SQL_MAX_TABLES_IN_SELECT
- SQL_MAX_USER_NAME_LEN

Scalar Function Information

The following values of *fInfoType* return information about the scalar functions that the data source and the driver support. For more information on scalar functions, refer to the *Informix Guide to SQL: Syntax*.

- SQL_NUMERIC_FUNCTIONS
- SQL_STRING_FUNCTIONS
- SQL_SYSTEM_FUNCTIONS
- SQL_TIMEDATE_ADD_INTERVALS
- SQL_TIMEDATE_DIFF_INTERVALS
- SQL_TIMEDATE_FUNCTIONS

Information-Type Descriptions

The following table lists each information type alphabetically with its description.

InfoType	Returns
SQL_ACCESSIBLE_PROCEDURES	A character string: ■ "Y" if the user can execute all procedures that **SQLProcedures** returns ■ "N" if procedures might be returned that the user cannot execute.
SQL_ACCESSIBLE_TABLES	A character string: ■ "Y" if the user is guaranteed SELECT privileges to all tables returned by **SQLTables** ■ "N" if tables might be returned that the user cannot access.
SQL_ACTIVE_CONNECTIONS	A 16-bit integer value that specifies the maximum number of active connection handles that the driver can support. This value can reflect a limitation that either the driver or the data source imposes. If no limit is specified or the limit is unknown, this value is set to 0.

(1 of 24)

InfoType	Returns
SQL_ACTIVE_STATEMENTS	A 16-bit integer value that specifies the maximum number of active *hstmts* that the driver can support for an *hdbc*. This value can reflect a limitation that either the driver or the data source imposes. If no limit is specified or the limit is unknown, this value is set to 0.
SQL_ALTER_TABLE	A 32-bit bitmask that enumerates the clauses in the ALTER TABLE statement supported by the data source. The following bitmask is used to determine which clauses are supported: SQL_AT_ADD_COLUMN SQL_AT_DROP_COLUMN
SQL_COLUMN_ALIAS	A character string: ■ "Y" if the data source supports column aliases; otherwise ■ "N" is returned.
SQL_CONCAT_NULL_BEHAVIOR	A 16-bit integer value that indicates how the data source handles the concatenation of null-valued character-data-type columns with non-null-valued character-data-type columns: ■ SQL_CB_NULL = Result is null valued. ■ SQL_CB_NON_NULL = Result is concatenation of non-null-valued column or columns.

(2 of 24)

InfoType	Returns
SQL_CONVERT_BIGINT SQL_CONVERT_BINARY SQL_CONVERT_BIT SQL_CONVERT_CHAR SQL_CONVERT_DATE SQL_CONVERT_DECIMAL SQL_CONVERT_DOUBLE SQL_CONVERT_FLOAT SQL_CONVERT_INTEGER SQL_CONVERT_LONGVARBINARY SQL_CONVERT_LONGVARCHAR SQL_CONVERT_NUMERIC SQL_CONVERT_REAL SQL_CONVERT_SMALLINT SQL_CONVERT_TIME SQL_CONVERT_TIMESTAMP SQL_CONVERT_TINYINT SQL_CONVERT_VARBINARY SQL_CONVERT_VARCHAR	A 32-bit bitmask. The bitmask indicates the conversions supported by the data source with the CONVERT scalar function for data of the type named in the *fInfoType*. If the bitmask equals 0, the data source does not support any conversions for data of the named type, including conversion to the same data type. For example, to find out if a data source supports the conversion of SQL_INTEGER data to the SQL_BIGINT data type, an application calls **SQLGetInfo** with the *fInfoType* of SQL_CONVERT_INTEGER. The application ANDs the returned bitmask with SQL_CVT_BIGINT. If the resulting value is not 0, the conversion is supported. The following bitmasks are used to determine which conversions are supported: SQL_CVT_BIGINT SQL_CVT_BINARY SQL_CVT_BIT SQL_CVT_CHAR SQL_CVT_DATE SQL_CVT_DECIMAL SQL_CVT_DOUBLE SQL_CVT_FLOAT SQL_CVT_INTEGER SQL_CVT_LONGVARBINARY SQL_CVT_LONGVARCHAR SQL_CVT_NUMERIC SQL_CVT_REAL SQL_CVT_SMALLINT SQL_CVT_TIME SQL_CVT_TIMESTAMP SQL_CVT_TINYINT SQL_CVT_VARBINARY SQL_CVT_VARCHAR
SQL_CONVERT_FUNCTIONS	A 32-bit bitmask that enumerates the scalar-conversion functions that the driver and associated data source support. The SQL_FN_CVT_CONVERT bitmask is used to determine which conversion functions are supported.

(3 of 24)

InfoType	Returns
SQL_CORRELATION_NAME	A 16-bit integer that indicates if table correlation names are supported: ■ SQL_CN_NONE = Correlation names are not supported. ■ SQL_CN_DIFFERENT = Correlation names are supported, but they must differ from the names of the tables that they represent. ■ SQL_CN_ANY = Correlation names are supported and can be any valid user-defined name.
SQL_CURSOR_COMMIT_BEHAVIOR	A 16-bit integer value that indicates how a COMMIT operation affects cursors and prepared statements in the data source: ■ SQL_CB_DELETE = Close cursors and delete prepared statements. To use the cursor again, the application must re-prepare and re-execute the *hstmt*. ■ SQL_CB_CLOSE = Close cursors. For prepared statements, the application can call **SQLExecute** on the *hstmt* without calling **SQLPrepare** again. ■ SQL_CB_PRESERVE = Preserve cursors in the same position as before the COMMIT operation. The application can continue to fetch data or it can close the cursor and re-execute the *hstmt* without re-preparing it.
SQL_CURSOR_ROLLBACK_BEHAVIOR	A 16-bit integer value that indicates how a ROLLBACK operation affects cursors and prepared statements in the data source: ■ SQL_CB_DELETE = Close cursors and delete prepared statements. To use the cursor again, the application must re-prepare and re-execute the *hstmt*. ■ SQL_CB_CLOSE = Close cursors. For prepared statements, the application can call **SQLExecute** on the *hstmt* without calling **SQLPrepare** again. ■ SQL_CB_PRESERVE = Preserve cursors in the same position as before the ROLLBACK operation. The application can continue to fetch data, or it can close the cursor and re-execute the *hstmt* without re-preparing it.

(4 of 24)

InfoType	**Returns**
SQL_DATA_SOURCE_NAME	A character string with the data-source name used during connection. If the application called **SQLConnect**, this is the value of the *szDSN* argument. If the application called **SQLDriverConnect** or **SQLBrowseConnect**, this is the value of the DSN keyword in the connection string passed to the driver. If the connection string did not contain the DSN keyword (as when it contains the DRIVER keyword), this is an empty string.
SQL_DATA_SOURCE_READ_ONLY	A character string: ■ "Y" if the data source is set to READ ONLY mode ■ "N" if it is otherwise. This characteristic pertains only to the data source; it is not a characteristic of the driver that enables access to the data source.
SQL_DATABASE_NAME	A character string with the name of the current database in use, if the data source defines a named object called "database."
SQL_DBMS_NAME	A character string with the name of the DBMS product accessed by the driver.
SQL_DBMS_VER	A character string that indicates the version of the DBMS product accessed by the driver. The version has the form ##.##.####, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version. The driver must render the DBMS product version in this form but can also append the DBMS product-specific version. For example, "04.01.0000 RDB 4.1."

(5 of 24)

InfoType	Returns
SQL_DEFAULT_TXN_ISOLATION	A 32-bit integer that indicates the default transaction isolation level supported by the driver or data source, or zero if the data source does not support transactions. The following terms are used to define transaction isolation levels: ■ *Dirty Read* Transaction 1 changes a row. Transaction 2 reads the changed row before transaction 1 commits the change. If transaction 1 rolls back the change, transaction 2 will have read a row that is considered to have never existed. ■ *Non-repeatable Read* Transaction 1 reads a row. Transaction 2 updates or deletes that row and commits this change. If transaction 1 attempts to reread the row, it will receive different row values or discover that the row has been deleted. ■ *Phantom* Transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 inserts a row that matches the search criteria. If transaction 1 re-executes the statement that read the rows, it receives a different set of rows. If the data source supports transactions, the driver returns one of the following bitmasks: ■ SQL_TXN_READ_UNCOMMITTED = Dirty Reads, Non-repeatable Reads, and Phantoms are possible. ■ SQL_TXN_READ_COMMITTED = Dirty Reads are not possible. Non-repeatable Reads and Phantoms are possible. ■ SQL_TXN_REPEATABLE_READ = Dirty Reads and Non-repeatable Reads are not possible. Phantoms are possible. ■ SQL_TXN_SERIALIZABLE = Transactions can be serialized. Dirty Reads, Non-repeatable Reads, and Phantoms are not possible. ■ SQL_TXN_VERSIONING = Transactions can be serialized, but higher concurrency is possible than with SQL_TXN_SERIALIZABLE. Dirty Reads are not possible. Typically, SQL_TXN_SERIALIZABLE is implemented by using locking protocols that reduce concurrency, and SQL_TXN_VERSIONING is implemented by using a non-locking protocol such as record versioning.
SQL_DRIVER_HDBC SQL_DRIVER_HENV	A 32-bit value, the environment handle or connection handle of the driver, determined by the argument *hdbc*.

(6 of 24)

InfoType	Returns
SQL_DRIVER_HLIB	A 32-bit value, the library handle returned to INFORMIX-CLI when it loaded the driver shared library. The handle is only valid for the connection handle (HDBC) specified in the call to **SQLGetInfo**.
SQL_DRIVER_HSTMT	A 32-bit value, the statement handle of the driver determined by the driver manager statement handle, which must be passed on input in *rgbInfoValue* from the application. In this case, *rgbInfoValue* is both an input and an output argument. The input *hstmt* passed in *rgbInfoValue* was probably an *hstmt* allocated on the argument *hdbc*.
SQL_DRIVER_NAME	A character string with the filename of the driver used to access the data source.
SQL_DRIVER_ODBC_VER	A character string with the version of ODBC that the driver supports. The version has the form ##.##, where the first two digits are the major version and the next two digits are the minor version. SQL_SPEC_MAJOR and SQL_SPEC_MINOR define the major and minor version numbers. For the version of ODBC described in this manual, these are `2` and `0`, and the driver should return `02.00`. If a driver supports **SQLGetInfo** but does not support this value of the *fInfoType* argument, the driver manager (if you are using one) returns `01.00`.
SQL_DRIVER_VER	A character string with the version of the driver and, optionally, a description of the driver. At a minimum, the version has the form ##.##.####, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version.
SQL_EXPRESSIONS_IN_ORDERBY	A character string: ■ “Y” if the data source supports expressions in the ORDER BY list ■ “N” if it does not.

(7 of 24)

InfoType	Returns
SQL_FETCH_DIRECTION	A 32-bit bitmask that enumerates the supported fetch direction options. The following bitmasks are used in conjunction with the flag to determine which options are supported: SQL_FD_FETCH_NEXT SQL_FD_FETCH_FIRST) SQL_FD_FETCH_LAST SQL_FD_FETCH_PRIOR SQL_FD_FETCH_ABSOLUTE SQL_FD_FETCH_RELATIVE SQL_FD_FETCH_RESUME
SQL_FILE_USAGE	A 16-bit integer value that indicates how a single-tier driver directly treats files in a data source: ■ SQL_FILE_NOT_SUPPORTED = The driver is not a single-tier driver. ■ SQL_FILE_TABLE = A single-tier driver treats files in a data source as tables. For example, a text driver treats each text file as a table. ■ SQL_FILE_QUALIFIER = A single-tier driver treats files in a data source as a qualifier.

(8 of 24)

InfoType	Returns
SQL_GETDATA_EXTENSIONS	A 32-bit bitmask that enumerates extensions to **SQLGetData**. The following bitmasks are used in conjunction with the flag to determine what common extensions the driver supports for **SQLGetData**: ■ SQL_GD_ANY_COLUMN = **SQLGetData** can be called for any unbound column, including those before the last bound column. The columns must be called in order of ascending column number unless SQL_GD_ANY_ORDER is also returned. ■ SQL_GD_ANY_ORDER = **SQLGetData** can be called for unbound columns in any order. **SQLGetData** can be called only for columns after the last bound column unless SQL_GD_ANY_COLUMN is also returned. ■ SQL_GD_BOUND = **SQLGetData** can be called for bound columns as well as unbound columns. A driver cannot return this value unless it also returns SQL_GD_ANY_COLUMN. **SQLGetData** is required to return data only from unbound columns that occur after the last bound column, are called in order of increasing column number, and are not in a row in a block of rows.

(9 of 24)

InfoType	Returns

SQL_GROUP_BY	A 16-bit integer value that specifies the relationship between the columns in the GROUP BY clause and the non-aggregated columns in the SELECT list:
	■ SQL_GB_NOT_SUPPORTED = GROUP BY clauses are not supported.
	■ SQL_GB_GROUP_BY_EQUALS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the SELECT list. It cannot contain any other columns. For example:
	`SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT`
	■ SQL_GB_GROUP_BY_CONTAINS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the SELECT list. It can contain columns that are not in the SELECT list. For example:
	`SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT, AGE`
	■ SQL_GB_NO_RELATION = The columns in the GROUP BY clause and the select list are not related. The meaning of non-grouped, non-aggregated columns in the SELECT list is data-source dependent. For example,
	`SElECT DEPT, SALARY FROM EMPLOYEE GROUP BY DEPT, AGE`

SQL_IDENTIFIER_CASE	A 16-bit integer value as follows: ■ SQL_IC_UPPER = Identifiers in SQL are case insensitive and are stored in uppercase in system catalog. ■ SQL_IC_LOWER = Identifiers in SQL are case insensitive and are stored in lowercase in system catalog. ■ SQL_IC_SENSITIVE = Identifiers in SQL are case sensitive and are stored in mixed case in system catalog. ■ SQL_IC_MIXED = Identifiers in SQL are case insensitive and are stored in mixed case in system catalog.
SQL_IDENTIFIER_QUOTE_CHAR	The character string used as the starting and ending delimiter of a quoted (delimited) identifiers in SQL statements. (Identifiers passed as arguments to ODBC functions do not need to be quoted.) If the data source does not support quoted identifiers, a blank is returned.

(10 of 24)

InfoType	Returns
SQL_KEYWORDS	A character string that contains a comma-separated list of all data-source-specific keywords. This list does not contain keywords specific to ODBC or keywords used by both the data source and ODBC. The #define value SQL_ODBC_KEYWORDS contains a comma-separated list of ODBC keywords.
SQL_LIKE_ESCAPE_CLAUSE	A character string: ■ "Y" if the data source supports an escape character for the percent character (%) and underscore character (_) in a LIKE predicate and the driver supports the ODBC syntax for defining a LIKE predicate escape character ■ "N" otherwise.
SQL_MAX_BINARY_LITERAL_LEN	A 32-bit integer value that specifies the maximum length (number of hexadecimal characters, excluding the literal prefix and suffix returned by **SQLGetTypeInfo**) of a binary literal in an SQL statement. For example, the binary literal 0xFFAA has a length of 4. If there is no maximum length or the length is unknown, this value is set to 0.
SQL_MAX_CHAR_LITERAL_LEN	A 32-bit integer value that specifies the maximum length (number of characters, excluding the literal prefix and suffix returned by **SQLGetTypeInfo**) of a character literal in an SQL statement. If there is no maximum length or the length is unknown, this value is set to 0.
SQL_MAX_COLUMN_NAME_LEN	A 16-bit integer value that specifies the maximum length of a column name in the data source. If there is no maximum length or the length is unknown, this value is set to 0.
SQL_MAX_COLUMNS_IN_GROUP_BY	A 16-bit integer value that specifies the maximum number of columns allowed in a GROUP BY clause. If no limit is specified or the limit is unknown, this value is set to 0.
SQL_MAX_COLUMNS_IN_INDEX	A 16-bit integer value that specifies the maximum number of columns allowed in an index. If no limit is specified or the limit is unknown, this value is set to 0.
SQL_MAX_COLUMNS_IN_ORDER_BY	A 16-bit integer value that specifies the maximum number of columns allowed in an ORDER BY clause. If no limit is specified or the limit is unknown, this value is set to 0.

(11 of 24)

InfoType	Returns
SQL_MAX_COLUMNS_IN_SELECT	A 16-bit integer value that specifies the maximum number of columns allowed in a SELECT list. If no limit is specified or the limit is unknown, this value is set to 0.
SQL_MAX_COLUMNS_IN_TABLE	A 16-bit integer value that specifies the maximum number of columns allowed in a table. If no limit is specified or the limit is unknown, this value is set to 0.
SQL_MAX_CURSOR_NAME_LEN	A 16-bit integer value that specifies the maximum length of a cursor name in the data source. If there is no maximum length or the length is unknown, this value is set to 0.
SQL_MAX_INDEX_SIZE	A 32-bit integer value that specifies the maximum number of bytes allowed in the combined fields of an index. If no limit is specified or the limit is unknown, this value is set to 0.
SQL_MAX_OWNER_NAME_LEN	A 16-bit integer value that specifies the maximum length of an owner name in the data source. If there is no maximum length or the length is unknown, this value is set to 0.
SQL_MAX_PROCEDURE_NAME_LEN	A 16-bit integer value that specifies the maximum length of a procedure name in the data source. If there is no maximum length or the length is unknown, this value is set to 0.
SQL_MAX_QUALIFIER_NAME_LEN	A 16-bit integer value that specifies the maximum length of a qualifier name in the data source. If there is no maximum length or the length is unknown, this value is set to 0.
SQL_MAX_ROW_SIZE	A 32-bit integer value that specifies the maximum length of a single row in a table. If no limit is specified or the limit is unknown, this value is set to 0.
SQL_MAX_ROW_SIZE_INCLUDES_LONG	A character string: ■ "Y" if the maximum row size returned for the SQL_MAX_ROW_SIZE information type includes the length of all SQL_LONGVARCHAR and SQL_LONGVARBINARY columns in the row ■ "N" otherwise.
SQL_MAX_STATEMENT_LEN	A 32-bit integer value that specifies the maximum length (number of characters, including white space) of an SQL statement. If there is no maximum length or the length is unknown, this value is set to 0.

(12 of 24)

InfoType	Returns
SQL_MAX_TABLE_NAME_LEN	A 16-bit integer value that specifies the maximum length of a table name in the data source. If there is no maximum length or the length is unknown, this value is set to 0.
SQL_MAX_TABLES_IN_SELECT	A 16-bit integer value that specifies the maximum number of tables allowed in the FROM clause of a SELECT statement. If there is no specified limit or the limit is unknown, this value is set to 0.
SQL_MAX_USER_NAME_LEN	A 16-bit integer value that specifies the maximum length of a user name in the data source. If there is no maximum length or the length is unknown, this value is set to 0.
SQL_MULT_RESULT_SETS	A character string: ■ "Y" if the data source supports multiple result sets ■ "N" if it does not.
SQL_MULTIPLE_ACTIVE_TXN	A character string: ■ "Y" if active transactions on multiple connections are allowed ■ "N" if only one connection at a time can have an active transaction.
SQL_NEED_LONG_DATA_LEN	A character string: ■ "Y" if the data source needs the length of a long data value (the data type is SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long, data-source-specific data type) before that value is sent to the data source ■ "N" if it does not. For more information, see **SQLBindParameter**.
SQL_NON_NULLABLE_COLUMNS	A 16-bit integer that specifies whether the data source supports non-nullable columns: ■ SQL_NNC_NULL = All columns must be nullable. ■ SQL_NNC_NON_NULL = Columns may be non-nullable (the data source supports the NOT NULL column constraint in CREATE TABLE statements).

(13 of 24)

InfoType	Returns
SQL_NULL_COLLATION	A 16-bit integer value that specifies where NULLs are sorted in a list: ■ SQL_NC_END = NULLs are sorted at the end of the list, regardless of the sort order. ■ SQL_NC_HIGH = NULLs are sorted at the high end of the list. ■ SQL_NC_LOW = NULLs are sorted at the low end of the list. ■ SQL_NC_START = NULLs are sorted at the start of the list, regardless of the sort order.
SQL_NUMERIC_FUNCTIONS	A 32-bit bitmask that enumerates the scalar numeric functions supported by the driver and associated data source. The following bitmasks are used to determine which numeric functions are supported: SQL_FN_NUM_ABS SQL_FN_NUM_ACOS SQL_FN_NUM_ASIN SQL_FN_NUM_ATAN SQL_FN_NUM_ATAN2 SQL_FN_NUM_CEILING SQL_FN_NUM_COS SQL_FN_NUM_COS SQL_FN_NUM_DEGREES SQL_FN_NUM_EXP SQL_FN_NUM_FLOOR SQL_FN_NUM_LOG SQL_FN_NUM_LOG10 SQL_FN_NUM_MOD SQL_FN_NUM_PI SQL_FN_NUM_POWER SQL_FN_NUM_RADIANS SQL_FN_NUM_RAND SQL_FN_NUM_ROUND SQL_FN_NUM_SIGN SQL_FN_NUM_SIN SQL_FN_NUM_SQRT SQL_FN_NUM_TAN SQL_FN_NUM_TRUNCATE

(14 of 24)

InfoType	Returns
SQL_ODBC_API_CONFORMANCE	A 16-bit integer value that indicates the level of ODBC conformance: ■ SQL_OAC_NONE = None ■ SQL_OAC_LEVEL1 = Level 1 supported ■ SQL_OAC_LEVEL2 = Level 2 supported For a list of functions and conformance levels, see Chapter 11, “Function Summary.”
SQL_ODBC_SAG_CLI_CONFORMANCE	A 16-bit integer value that indicates compliance to the functions of the SAG specification: ■ SQL_OSCC_NOT_COMPLIANT = Not SAG-compliant; one or more core functions are not supported. ■ SQL_OSCC_COMPLIANT = SAG-compliant
SQL_ODBC_SQL_CONFORMANCE	A 16-bit integer value that indicates SQL grammar supported by the driver: ■ SQL_OSC_MINIMUM = Minimum grammar supported ■ SQL_OSC_CORE = Core grammar supported ■ SQL_OSC_EXTENDED = Extended grammar supported
SQL_ODBC_SQL_OPT_IEF	A character string: ■ “Y” if the data source supports the optional Integrity Enhancement Facility ■ “N” if it does not
SQL_ODBC_VER	A character string with the version of ODBC to which the driver manager conforms, if you are using a driver manager. The version is of the form ##.##, where the first two digits are the major version and the next two digits are the minor version.
SQL_ORDER_BY_COLUMNS_IN_SELECT	A character string: ■ “Y” if the columns in the ORDER BY clause must be in the SELECT list ■ “N” otherwise

(15 of 24)

InfoType	Returns
SQL_OUTER_JOINS	A character string: ■ "N" = No. The data source does not support outer joins. ■ "Y" = Yes. The data source supports two-table outer joins, and the driver supports the ODBC outer join syntax except for nested outer joins. However, columns on the left side of the comparison operator in the ON clause must come from the left-hand table in the outer join, and columns on the right side of the comparison operator must come from the right-hand table. ■ "P" = Partial. The data source partially supports nested outer joins, and the driver supports the ODBC outer-join syntax. However, columns on the left side of the comparison operator in the ON clause must come from the left-hand table in the outer join and columns on the right side of the comparison operator must come from the right-hand table. Also, the right-hand table of an outer join cannot be included in an inner join. ■ "F" = Full. The data source fully supports nested outer joins, and the driver supports the ODBC outer-join syntax.
SQL_OWNER_TERM	A character string with the data-source vendor name for an owner; for example, "owner," "Authorization ID," or "Schema."
SQL_OWNER_USAGE	A 32-bit bitmask that enumerates the statements in which owners can be used: ■ SQL_OU_DML_STATEMENTS = Owners are supported in all Data Manipulation Language statements: SELECT, INSERT, UPDATE, DELETE, and, if supported, SELECT FOR UPDATE and positioned UPDATE and DELETE statements. ■ SQL_OU_PROCEDURE_INVOCATION = Owners are supported in the ODBC procedure invocation statement. ■ SQL_OU_TABLE_DEFINITION = Owners are supported in all table definition statements: CREATE TABLE, CREATE VIEW, ALTER TABLE, DROP TABLE, and DROP VIEW. ■ SQL_OU_INDEX_DEFINITION = Owners are supported in all index-definition statements: CREATE INDEX and DROP INDEX. ■ SQL_OU_PRIVILEGE_DEFINITION = Owners are supported in all privilege-definition statements: GRANT and REVOKE.

(16 of 24)

InfoType	Returns
SQL_POSITIONED_STATEMENTS	A 32-bit bitmask that enumerates the supported positioned SQL statements. The following bitmasks are used to determine which statements are supported: SQL_PS_POSITIONED_DELETE SQL_PS_POSITIONED_UPDATE SQL_PS_SELECT_FOR_UPDATE
SQL_PROCEDURE_TERM	A character string with the data-source vendor name for a procedure; for example, "database procedure," "stored procedure," or "procedure."
SQL_PROCEDURES	A character string: ■ "Y" if the data source supports procedures and the driver supports the ODBC procedure invocation syntax ■ "N" otherwise.
SQL_QUALIFIER_LOCATION	A 16-bit integer value that indicates the position of the qualifier in a qualified table name: SQL_QL_START SQL_QL_END For example, a Text driver returns SQL_QL_START because the directory (qualifier) name is at the start of the table name, as in **/empdata/emp.dbf**.
SQL_QUALIFIER_NAME_SEPARATOR	A character string: the character or characters that the data source defines as the separator between a qualifier name and the qualified name element that follows it.
SQL_QUALIFIER_TERM	A character string with the data-source vendor name for a qualifier; for example, "database" or "directory."

(17 of 24)

InfoType	Returns
SQL_QUALIFIER_USAGE	A 32-bit bitmask that enumerates the statements in which qualifiers can be used. The following bitmasks are used to determine where qualifiers can be used: ■ SQL_QU_DML_STATEMENTS = Qualifiers are supported in all Data Manipulation Language statements: SELECT, INSERT, UPDATE, DELETE, and, if supported, SELECT FOR UPDATE and positioned UPDATE and DELETE statements. ■ SQL_QU_PROCEDURE_INVOCATION = Qualifiers are supported in the ODBC procedure invocation statement. ■ SQL_QU_TABLE_DEFINITION = Qualifiers are supported in all table-definition statements: CREATE TABLE, CREATE VIEW, ALTER TABLE, DROP TABLE, and DROP VIEW. ■ SQL_QU_INDEX_DEFINITION = Qualifiers are supported in all index-definition statements: CREATE INDEX and DROP INDEX. ■ SQL_QU_PRIVILEGE_DEFINITION = Qualifiers are supported in all privilege-definition statements: GRANT and REVOKE.
SQL_QUOTED_IDENTIFIER_CASE	A 16-bit integer value as follows: ■ SQL_IC_UPPER = Quoted identifiers in SQL are case insensitive and are stored in uppercase in system catalog. ■ SQL_IC_LOWER = Quoted identifiers in SQL are case insensitive and are stored in lowercase in system catalog. ■ SQL_IC_SENSITIVE = Quoted identifiers in SQL are case sensitive and are stored in mixed case in system catalog. ■ SQL_IC_MIXED = Quoted identifiers in SQL are not case sensitive and are stored in mixed case in system catalog.
SQL_ROW_UPDATES	A character string: ■ "Y" if a key-set-driven or mixed cursor maintains row versions or values for all fetched rows and therefore can detect any changes made to a row by any user since the row was last fetched ■ "N" otherwise.

(18 of 24)

InfoType	**Returns**
SQL_SCROLL_CONCURRENCY	A 32-bit bitmask that enumerates the concurrency control options supported for scrollable cursors. The following bitmasks are used to determine which options are supported: ■ SQL_SCCO_READ_ONLY = Cursor is read only. No updates are allowed. ■ SQL_SCCO_LOCK = Cursor uses the lowest level of locking sufficient to ensure that the row can be updated. ■ SQL_SCCO_OPT_ROWVER = Cursor uses optimistic concurrency control, comparing row versions. ■ SQL_SCCO_OPT_VALUES = Cursor uses optimistic concurrency control, comparing values. For information about cursor concurrency, see "Specifying Cursor Concurrency" on page 6-11."
SQL_SCROLL_OPTIONS	A 32-bit bitmask that enumerates the scroll options supported for scrollable cursors. The following bitmasks are used to determine which options are supported: ■ SQL_SO_FORWARD_ONLY = The cursor only scrolls forward. ■ SQL_SO_STATIC = The data in the result set is static. ■ SQL_SO_KEYSET_DRIVEN = The driver saves and uses the keys for every row in the result set. ■ SQL_SO_DYNAMIC = The driver keeps the keys for every row in the row set (the key-set size is the same as the row-set size). ■ SQL_SO_MIXED = The driver keeps the keys for every row in the key set, and the key-set size is greater than the row-set size. The cursor is key-set driven inside the key set and dynamic outside the key set. For information about scrollable cursors, see"Scrollable Cursors" on page 6-10.

(19 of 24)

InfoType	Returns
SQL_SEARCH_PATTERN_ESCAPE	A character string that specifies what the driver supports as an escape character that permits the use of the pattern-match metacharacters underscore (_) and percent (%) as valid characters in search patterns. This escape character applies only for those catalog function arguments that support search strings. If this string is empty, the driver does not support a search-pattern escape character. This *fInfoType* is limited to catalog functions. For a description of the use of the escape character in search pattern strings, see "Search Pattern Arguments" on page 12-8.
SQL_SERVER_NAME	A character string with the actual data-source-specific server name; useful when a data-source name is used during **SQLConnect, SQLDriverConnect,** and **SQLBrowseConnect**.
SQL_SPECIAL_CHARACTERS	A character string that contains all special characters (that is, all characters except a through z, A through Z, 0 through 9, and underscore) that can be used in an object name, such as a table, column, or index name, on the data source. For example, "#$^."

(20 of 24)

InfoType	Returns
SQL_STATIC_SENSITIVITY	A 32-bit bitmask that enumerates whether changes made by an application to a static or key-set-driven cursor through positioned update or delete statements can be detected by that application: ■ SQL_SS_ADDITIONS = Added rows are visible to the cursor; the cursor can scroll to these rows. Where these rows are added to the cursor is driver dependent. ■ SQL_SS_DELETIONS = Deleted rows are no longer available to the cursor and do not leave a "hole" in the result set; after the cursor scrolls from a deleted row, it cannot return to that row. ■ SQL_SS_UPDATES = Updates to rows are visible to the cursor; if the cursor scrolls from and returns to an updated row, the data returned by the cursor is the updated data, not the original data. Because updating key values in a key-set-driven cursor is considered to be deleting the existing row and adding a new row, this value is always returned for key-set-driven cursors. Whether an application can detect changes made to the result set by other users, including other cursors in the same application, depends on the cursor type. For more information, see"Scrollable Cursors" on page 6-10.
SQL_SYSTEM_FUNCTIONS	A 32-bit bitmask that enumerates the scalar system functions supported by the driver and associated data source. The following bitmasks are used to determine which system functions are supported: SQL_FN_SYS_DBNAME SQL_FN_SYS_IFNULL SQL_FN_SYS_USERNAME
SQL_TABLE_TERM	A character string with the data-source vendor name for a table; for example, "table" or "file."

(21 of 24)

InfoType	Returns
SQL_TIMEDATE_ADD_INTERVALS	A 32-bit bitmask that enumerates the time-stamp intervals supported by the driver and associated data source for the TIMESTAMPADD scalar function. The following bitmasks are used to determine which intervals are supported: SQL_FN_TSI_FRAC_SECOND SQL_FN_TSI_SECOND SQL_FN_TSI_MINUTE SQL_FN_TSI_HOUR SQL_FN_TSI_DAY SQL_FN_TSI_WEEK SQL_FN_TSI_MONTH SQL_FN_TSI_QUARTER SQL_FN_TSI_YEAR
SQL_TIMEDATE_DIFF_INTERVALS	A 32-bit bitmask that enumerates the time-stamp intervals supported by the driver and associated data source for the TIMESTAMPDIFF scalar function. The following bitmasks are used to determine which intervals are supported: SQL_FN_TSI_FRAC_SECOND SQL_FN_TSI_SECOND SQL_FN_TSI_MINUTE SQL_FN_TSI_HOUR SQL_FN_TSI_DAY SQL_FN_TSI_WEEK SQL_FN_TSI_MONTH SQL_FN_TSI_QUARTER SQL_FN_TSI_YEAR

(22 of 24)

InfoType	Returns
SQL_TIMEDATE_FUNCTIONS	A 32-bit bitmask that enumerates the scalar date and time functions supported by the driver and associated data source. The following bitmasks are used to determine which date and time functions are supported: SQL_FN_TD_CURDATE SQL_FN_TD_CURTIME SQL_FN_TD_DAYNAME SQL_FN_TD_DAYOFMONTH SQL_FN_TD_DAYOFWEEK SQL_FN_TD_DAYOFYEAR SQL_FN_TD_HOUR SQL_FN_TD_MINUTE SQL_FN_TD_MONTH SQL_FN_TD_MONTHNAME SQL_FN_TD_NOW SQL_FN_TD_QUARTER SQL_FN_TD_SECOND SQL_FN_TD_TIMESTAMPADD SQL_FN_TD_TIMESTAMPDIFF SQL_FN_TD_WEEK SQL_FN_TD_YEAR
SQL_TXN_CAPABLE	A 16-bit integer value that describes the transaction support in the driver or data source: ■ SQL_TC_NONE = Transactions not supported. ■ SQL_TC_DML = Transactions can contain only Data Manipulation Language (DML) statements (SELECT, INSERT, UPDATE, DELETE). Data Definition Language (DDL) statements encountered in a transaction cause an error. ■ SQL_TC_DDL_COMMIT = Transactions can contain only DML statements. DDL statements (CREATE TABLE, DROP INDEX, an so on) encountered in a transaction cause the transaction to be committed. ■ SQL_TC_DDL_IGNORE = Transactions can contain only DML statements. DDL statements encountered in a transaction are ignored. ■ SQL_TC_ALL = Transactions can contain DDL statements and DML statements in any order.

(23 of 24)

InfoType	Returns
SQL_TXN_ISOLATION_OPTION	A 32-bit bitmask that enumerates the transaction-isolation levels available from the driver or data source. The following bitmasks are used in conjunction with the flag to determine which options are supported: SQL_TXN_READ_UNCOMMITTED SQL_TXN_READ_COMMITTED SQL_TXN_REPEATABLE_READ SQL_TXN_SERIALIZABLE SQL_TXN_VERSIONING For descriptions of these isolation levels, see "SQL_DEFAULT_TXN_ISOLATION" on page 12-198.
SQL_UNION	A 32-bit bitmask that enumerates the support for the UNION clause: ■ SQL_U_UNION = The data source supports the UNION clause. ■ SQL_U_UNION_ALL = The data source supports the ALL keyword in the UNION clause. (**SQLGetInfo** returns both SQL_U_UNION and SQL_U_UNION_ALL in this case.)
SQL_USER_NAME	A character string with the name used in a particular database, which can be different than login name.

(24 of 24)

Code Example

SQLGetInfo returns lists of supported options as a 32-bit bitmask in *rgbInfoValue*. The bitmask for each option is used with the flag to determine whether the option is supported.

For example, an application could use the following code to determine whether the driver associated with the *hdbc* supports the SUBSTRING scalar function:

```
UDWORD fFuncs;

SQLGetInfo(hdbc,
           SQL_STRING_FUNCTIONS,
           (PTR)&fFuncs,
           sizeof(fFuncs),
           NULL);

if (fFuncs & SQL_FN_STR_SUBSTRING) /* SUBSTRING supported */
    ...;
else                               /* SUBSTRING not supported */
    ...;
```

Related Functions

For Information About	See
Returning the setting of a connection option	**SQLGetConnectOption**
Determining if a driver supports a function	**SQLGetFunctions**
Returning the setting of a statement option	**SQLGetStmtOption**
Returning information about the data types of a data source	**SQLGetTypeInfo**

♦

Level 1

SQLGetStmtOption

SQLGetStmtOption returns the current setting of a statement option.

Syntax

```
   RETCODE SQLGetStmtOption(hstmt, fOption, pvParam)
```

The **SQLGetStmtOption** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UWORD	*fOption*	Input	Option to retrieve.
PTR	*pvParam*	Output	Value associated with *fOption*. Depending on the value of *fOption*, a 32-bit integer value or a pointer to a null-terminated character string will be returned in *pvParam*.
SQL_GET_ROWID		Output	Returns the row ID of the last row inserted; returns 0 if the last SQL operation was not Insert.
SQL-GET_SERIAL _VALUE		Output	Returns the serial ID of the last row inserted; returns 0 if the last SQL operation was not Insert.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state	The argument *fOption* was SQL_ROW_NUMBER, but the cursor was not open, or the cursor was positioned before the start of the result set or after the end of the result set.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message returned by **SQLError** in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1011	Operation invalid at this time	The *fOption* argument was SQL_GET_BOOKMARK, and the value of the SQL_USE_BOOKMARKS statement option was SQL_UB_OFF.

(1 of 2)

SQLSTATE	Error	Description
S1092	Option type out of range	(DM) The value specified for the argument *fOption* was in the block of numbers reserved for ODBC connection and statement options but was not valid for the version of ODBC supported by the driver.
S1109	Invalid cursor position	The *fOption* argument was SQL_ROW_NUMBER, but the value in the *rgfRowStatus* array in **SQLExtendedFetch** for the current row was SQL_ROW_DELETED or SQL_ROW_ERROR.
S1C00	Driver not capable	The driver or data source does not support the value specified for the argument *f*Option.

(2 of 2)

Usage

The following table lists statement options for which corresponding values can be returned but not set. For a list of options that can be set and retrieved, see "Usage" on page 12-298. If *fOption* specifies an option that returns a string, *pvParam* must be a pointer to storage for the string. The maximum length of the string is SQL_MAX_OPTION_STRING_LENGTH bytes, excluding the null-termination byte.

fOption	pvParam contents
SQL_GET_ROWID	Returns the row ID of the last row inserted; returns 0 if the last SQL operation was not Insert.
SQL-GET_SERIAL_VALUE	Returns the serial ID of the last row inserted; returns 0 if the last SQL operation was not Insert.
SQL_ROW_NUMBER	A 32-bit integer value that specifies the number of the current row in the entire result set. If the number of the current row cannot be determined or there is no current row, the driver returns 0.

Related Functions

For Information About	See
Returning the setting of a connection option	**SQLGetConnectOption**
Setting a connection option	**SQLSetConnectOption**
Setting a statement option	**SQLSetStmtOption**

♦

Level 1

SQLGetTypeInfo

SQLGetTypeInfo returns information about data types that the data source supports. The driver returns the information in the form of an SQL result set.

Important: *Applications must use the type names returned in the TYPE_NAME column in ALTER TABLE and CREATE TABLE statements; they must not use the sample type names listed in Appendix C, "Comparison of INFORMIX-CLI and Embedded SQL."* ***SQLGetTypeInfo*** *might return more than one row with the same value in the DATA_TYPE column.*

Syntax

```
    RETCODE SQLGetTypeInfo(hstmt, fSqlType)
```

The **SQLGetTypeInfo** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle for the result set.
SWORD	*fSqlType*	Input	The parameter's INFORMIX-CLI SQL data type. The possible values are any of the *fSqlType* values listed in Appendix B, "Data Types," as well as SQL_ALL_TYPES which specifies that information about all data types should be returned. For more information about data types and conversions, see Appendix B.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1004	SQL data type out of range	(DM) The value specified for the argument *fSqlType* was in the block of numbers reserved for ODBC SQL data-type indicators but was not a valid ODBC SQL data-type indicator.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.

(1 of 2)

SQLSTATE	Error	Description
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1C00	Driver not capable	The driver or data source does not support the value specified for the argument *fSqlType*. The combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options was not supported by the driver or data source.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

SQLGetTypeInfo returns the results as a standard result set, ordered by DATA_TYPE and TYPE_NAME. The following table lists the columns in the result set.

The lengths of VARCHAR columns shown in the following table are maximums; the actual lengths depend on the data source.

Column Name	Data Type	Comments
TYPE_NAME	VARCHAR(128) not NULL	Data-source-dependent data-type name; for example, CHAR, VARCHAR, MONEY, LONG VARBINARY, or CHAR () FOR BIT DATA. Applications must use this name in CREATE TABLE and ALTER TABLE statements.
DATA_TYPE	SMALLINT not NULL	Informix SQL data type. The possible values are any of the Informix SQL data types listed in Appendix B, "Data Types." For more information about data types and conversions, see Appendix B.
PRECISION	INTEGER	The maximum precision of the data type on the data source. Null is returned for data types where precision is not applicable. For more information on precision, see "Precision, Scale, Length, and Display Size" on page B-5.
LITERAL_PREFIX	VARCHAR(128)	Character or characters used to prefix a literal; for example, a single quote (') for character data types or `0x` for binary data types; null is returned for data types where a literal prefix is not applicable.
LITERAL_SUFFIX	VARCHAR(128)	Character or characters used to terminate a literal; for example, a single quote (') for character data types; null is returned for data types where a literal suffix is not applicable.

(1 of 4)

Column Name	Data Type	Comments
CREATE_PARAMS	VARCHAR(128)	Parameters for a data-type definition. For example, CREATE_PARAMS for DECIMAL would be "precision, scale;" CREATE_PARAMS for VARCHAR would equal "max length;" null is returned if no parameters for the data type definition exist; for example, INTEGER. The driver supplies the CREATE_PARAMS text in the language of the country where it is used.
NULLABLE	SMALLINT not NULL	Whether the data type accepts a null value: ■ SQL_NO_NULLS if the data type does not accept null values. ■ SQL_NULLABLE if the data type accepts null values. ■ SQL_NULLABLE_UNKNOWN if it is not known if the column accepts null values.
CASE_SENSITIVE	SMALLINT not NULL	Whether a character data type is case sensitive in collations and comparisons: ■ TRUE if the data type is a character data type and is case sensitive. ■ FALSE if the data type is not a character data type or is not case sensitive.

(2 of 4)

Column Name	Data Type	Comments
SEARCHABLE	SMALLINT not NULL	How the data type is used in a WHERE clause: ■ SQL_UNSEARCHABLE if the data type cannot be used in a WHERE clause. ■ SQL_LIKE_ONLY if the data type can be used in a WHERE clause only with the LIKE predicate. ■ SQL_ALL_EXCEPT_LIKE if the data type can be used in a WHERE clause with all comparison operators except LIKE. ■ SQL_SEARCHABLE if the data type can be used in a WHERE clause with any comparison operator.
UNSIGNED_ ATTRIBUTE	SMALLINT	Whether the data type is unsigned: ■ TRUE if the data type is unsigned. ■ FALSE if the data type is signed. ■ null is returned if the attribute is not applicable to the data type or the data type is not numeric.
MONEY	SMALLINT not NULL	Whether the data type is a money data type: ■ TRUE if it is a money data type. ■ FALSE if it is not.

(3 of 4)

Column Name	Data Type	Comments
AUTO_INCREMENT	SMALLINT	Whether the data type is auto-incrementing: ■ TRUE if the data type is auto-incrementing. ■ FALSE if the data type is not auto-incrementing. ■ null is returned if the attribute is not applicable to the data type or the data type is not numeric. An application can insert values into a column having this attribute, but cannot update the values in the column.
LOCAL_TYPE_NAME	VARCHAR(128)	Localized version of the data-source-dependent name of the data type. Null is returned if a localized name is not supported by the data source. This name is intended for display only, as in dialog boxes.
MINIMUM_SCALE	SMALLINT	The minimum scale of the data type on the data source. If a data type has a fixed scale, the MINIMUM_SCALE and MAXIMUM_SCALE columns both contain this value. For example, an SQL_TIMESTAMP column might have a fixed scale for fractional seconds. Null is returned where scale is not applicable. For more information, see "Precision, Scale, Length, and Display Size" on page B-5.
MAXIMUM_SCALE	SMALLINT	The maximum scale of the data type on the data source. Null is returned where scale is not applicable. If the maximum scale is not defined separately on the data source but is instead defined to be the same as the maximum precision, this column contains the same value as the PRECISION column. For more information, see "Precision, Scale, Length, and Display Size" on page B-5.

(4 of 4)

Attribute information can apply to data types or to specific columns in a result set. **SQLGetTypeInfo** returns information about attributes associated with data types; **SQLColAttributes** returns information about attributes associated with columns in a result set.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning information about a column in a result set	**SQLColAttributes**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning information about a driver or data source	**SQLGetInfo**

♦

Level 2

SQLMoreResults

SQLMoreResults determines whether more results are available on an *hstmt* that contains SELECT, UPDATE, INSERT, or DELETE statements and, if so, initializes processing for those results.

Syntax

```
    RETCODE SQLMoreResults(hstmt)
```

The **SQLMoreResults** function accepts the following argument.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver was unable to allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

Usage

SELECT statements return result sets. UPDATE, INSERT, and DELETE statements return a count of affected rows. If any of these statements are batched, submitted with arrays of parameters, or in procedures, they can return multiple result sets or counts.

If another result set or count is available, **SQLMoreResults** returns SQL_SUCCESS and initializes the result set or count for additional processing. After an application calls **SQLMoreResults** for SELECT statements, it can call functions to determine the characteristics of the result set and to retrieve data from the result set. After calling **SQLMoreResults** for UPDATE, INSERT, or DELETE statements, an application can call **SQLRowCount**.

If all results have been processed, **SQLMoreResults** returns SQL_NO_DATA_FOUND.

If a current result set has unfetched rows, **SQLMoreResults** discards that result set and makes the next result set or count available.

If a batch of statements or a procedure mixes other SQL statements with SELECT, UPDATE, INSERT, and DELETE statements, these other statements do not affect **SQLMoreResults**.

For additional information about the valid sequencing of result-processing functions, see Appendix B, "ODBC State Transition Tables."

Related Functions

For Information About	See
Canceling statement processing	**SQLCancel**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Fetching part or all of a column of data	**SQLGetData**

♦

Level 2

SQLNativeSql

SQLNativeSql returns the SQL string that the driver translates.

Syntax

```
RETCODE SQLNativeSql(hdbc, szSqlStrIn, cbSqlStrIn, szSqlStr,
cbSqlStrMax, pcbSqlStr)
```

The **SQLNativeSql** function accepts the following arguments.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.
UCHAR FAR *	*szSqlStrIn*	Input	SQL text string to be translated.
SDWORD	*cbSqlStrIn*	Input	Length of *szSqlStrIn* text string.
UCHAR FAR *	*szSqlStr*	Output	Pointer to storage for the translated SQL string.
SDWORD	*cbSqlStrMax*	Input	Maximum length of the *szSqlStr* buffer.
SDWORD FAR*	*pcbSqlStr*	Output	The total number of bytes (excluding the null-termination byte) available to return in *szSqlStr*. If the number of bytes available to return is greater than or equal to *cbSqlStrMax*, the translated SQL string in *szSqlStr* is truncated to *cbSqlStrMax* – 1 bytes.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The buffer *szSqlStr* was not large enough to return the entire SQL string, so the SQL string was truncated. The argument *pcbSqlStr* contains the length of the untruncated SQL string (function returns SQL_SUCCESS_WITH_INFO).
08003	Connection not open	The *hdbc* was not in a connected state.
37000	Syntax error or access violation	The argument *szSqlStrIn* contained an SQL statement that was not preparable or contained a syntax error.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.

(1 of 2)

SQLSTATE	Error	Description
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1009	Invalid argument value	(DM) The argument *szSqlStrIn* was a null pointer.
S1090	Invalid string or buffer length	(DM) The argument *cbSqlStrIn* was less than 0 but not equal to SQL_NTS.
		(DM) The argument *cbSqlStrMax* was less than 0, and the argument *szSqlStr* was not a null pointer.

(2 of 2)

Usage

The following example shows what **SQLNativeSql** might return for an input SQL string that contains the scalar function LENGTH:

```
SELECT {fn LENGTH(NAME)} FROM EMPLOYEE
```

An INFORMIX-CLI driver might return the following translated SQL string:

```
SELECT length(NAME) FROM EMPLOYEE
```

Related Functions

None.

♦

Level 2

SQLNumParams

SQLNumParams returns the number of parameters in an SQL statement.

Syntax

```
    RETCODE SQLNumParams(hstmt, pcpar)
```

The **SQLNumParams** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle
SWORD FAR *	*pcpar*	Output	Number of parameters in the statement

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1010	Function-sequence error	(DM) The function was called prior to calling **SQLPrepare** or **SQLExecDirect** for the *hstmt*. (DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

Usage

SQLNumParams can be called only after **SQLPrepare** is called.

If the statement associated with *hstmt* does not contain parameters, **SQLNumParams** sets *pcpar* to 0.

Related Functions

For Information About	See
Returning information about a parameter in a statement	**SQLDescribeParam** (extension)
Assigning storage for a parameter	**SQLBindParameter**

♦

Core

SQLNumResultCols

SQLNumResultCols returns the number of columns in a result set.

Syntax

```
    RETCODE SQLNumResultCols(hstmt, pccol)
```

The **SQLNumResultCols** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle
SWORD FAR *	*pccol*	Output	Number of columns in the result set

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver was unable to allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1010	Function-sequence error	(DM) The function was called prior to calling **SQLPrepare** or **SQLExecDirect** for the *hstmt*. (DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

SQLNumResultCols can return any SQLSTATE that **SQLPrepare** or **SQLExecute** can return when **SQLNumResultCols** is called after **SQLPrepare** and before **SQLExecute** is called, depending on when the data source evaluates the SQL statement associated with the *hstmt*.

Usage

SQLNumResultCols can be called successfully only when the *hstmt* is in the prepared, executed, or positioned state.

If the statement associated with *hstmt* does not return columns, **SQLNumResultCols** sets *pccol* to 0.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning information about a column in a result set	**SQLColAttributes**
Returning information about a column in a result set	**SQLDescribeCol**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Fetching part or all of a column of data	**SQLGetData**
Setting cursor scrolling options	**SQLSetScrollOptions**

♦

Level 1

SQLParamData

SQLParamData is used with **SQLPutData** to supply parameter data when a statement executes.

Syntax

```
RETCODE SQLParamData(hstmt, prgbValue)
```

The **SQLParamData** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
PTR FAR *	*prgbValue*	Output	Pointer to storage for the value specified for the *rgbValue* argument in **SQLBindParameter** (for parameter data) or the address of the *rgbValue* buffer specified in **SQLBindCol** (for column data).

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLEINFORMIX-CLI

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
22026	String data, length mismatch	The SQL_NEED_LONG_DATA_LEN information type in **SQLGetInfo** was “Y”, and less data was sent for a long parameter (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long, data-source-specific data type) than was specified with the *pcbValue* argument in **SQLBindParameter**.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.

(1 of 2)

SQLSTATE	Error	Description
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application. **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. **SQLCancel** was called before data was sent for all data-at-execution parameters or columns.
S1010	Function-sequence error	(DM) The previous function call was not a call to **SQLExecDirect** or **SQLExecute** where the return code was SQL_NEED_DATA or a call to **SQLPutData**. The previous function call was a call to **SQLParamData**.
S1T00	Time-out expired	The time-out period expired before the data source completed processing the parameter value. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

If **SQLParamData** is called while sending data for a parameter in an SQL statement, it can return any SQLSTATE that can be returned by the function that was called to execute the statement (**SQLExecute** or **SQLExecDirect**).

Usage

For an explanation of how data-at-execution parameter data is passed at statement-execution time, see "Passing Parameter Values" on page 12-33.

Code Example

See **SQLPutData**.

Related Functions

For Information About	See
Canceling statement processing	**SQLCancel**
Returning information about a parameter in a statement	**SQLDescribeParam** (extension)
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Sending parameter data at execution time	**SQLPutData**
Assigning storage for a parameter	**SQLBindParameter**

♦

Core

SQLPrepare

SQLPrepare prepares an SQL string for execution.

Syntax

```
    RETCODE SQLPrepare(hstmt, szSqlStr, cbSqlStr)
```

The **SQLPrepare** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle
UCHAR FAR *	*szSqlStr*	Input	SQL text string
SDWORD	*cbSqlStr*	Input	Length of *szSqlStr*

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
21S01	Insert value list does not match column list.	The argument *szSqlStr* contained an INSERT statement, and the number of values to be inserted did not match the degree of the derived table.
21S02	Degree of derived table does not match column list.	The argument *szSqlStr* contained a CREATE VIEW statement, and the number of names specified is not the same degree as the derived table that the query specification defines.
22005	Error in assignment	The argument *szSqlStr* contained an SQL statement that contained a literal or parameter, and the value was incompatible with the data type of the associated table column.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
34000	Invalid cursor name	The argument *szSqlStr* contained a positioned DELETE or a positioned UPDATE, and the cursor referenced by the statement being prepared was not open.

(1 of 4)

SQLSTATE	Error	Description
37000	Syntax error or access violation	The argument *szSqlStr* contained an SQL statement that was not preparable or contained a syntax error.
42000	Syntax error or access violation	The argument *szSqlStr* contained a statement for which the user did not have the required privileges.
S0001	Base table or view already exists	The argument *szSqlStr* contained a CREATE TABLE or CREATE VIEW statement, but the table name or view name specified already exists.
S0002	Base table not found	The argument *szSqlStr* contained a DROP TABLE or a DROP VIEW statement, but the specified table name or view name did not exist. The argument *szSqlStr* contained an ALTER TABLE statement, but the specified table name did not exist. The argument *szSqlStr* contained a CREATE VIEW statement, but a table name or view name that the query specification defined did not exist. The argument *szSqlStr* contained a CREATE INDEX statement, but the specified table name did not exist. The argument *szSqlStr* contained a GRANT or REVOKE statement, but the specified table name or view name did not exist. The argument *szSqlStr* contained a SELECT statement, but a specified table name or view name did not exist. The argument *szSqlStr* contained a DELETE, INSERT, or UPDATE statement, but the specified table name did not exist. The argument *szSqlStr* contained a CREATE TABLE statement, and a table specified in a constraint (referencing a table other than the one being created) did not exist.

(2 of 4)

SQLSTATE	Error	Description
S0011	Index already exists.	The argument *szSqlStr* contained a CREATE INDEX statement, but the specified index name already existed.
S0012	Index not found	The argument *szSqlStr* contained a DROP INDEX statement, but the specified index name did not exist.
S0021	Column already exists.	The argument *szSqlStr* contained an ALTER TABLE statement, and the column specified in the ADD clause is not unique or identifies an existing column in the base table.
S0022	Column not found	The argument *szSqlStr* contained a CREATE INDEX statement, and one or more of the column names specified in the column list did not exist. The argument *szSqlStr* contained a GRANT or REVOKE statement, and a specified column name did not exist. The argument *szSqlStr* contained a SELECT, DELETE, INSERT, or UPDATE statement, and a specified column name did not exist. The argument *szSqlStr* contained a CREATE TABLE statement, and a column specified in a constraint (referencing a table other than the one being created) did not exist.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1009	Invalid argument value	(DM) The argument *szSqlStr* was a null pointer.

(3 of 4)

SQLSTATE	Error	Description
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The argument *cbSqlStr* was less than or equal to 0, but not equal to SQL_NTS.
S1C00	Driver not capable	The cursor/concurrency combination is invalid.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(4 of 4)

Usage

The application calls **SQLPrepare** to send an SQL statement to the data source for preparation. The application can include one or more parameter markers in the SQL statement. To include a parameter marker, the application embeds a question mark (?) into the SQL string at the appropriate position.

Tip: *If an application uses **SQLPrepare** to prepare and **SQLExecute** to submit a COMMIT or ROLLBACK statement, it is not interoperable among DBMS products. To commit or roll back a transaction, call **SQLTransact**.*

The driver modifies the statement to use the form of SQL that the data source uses and then submits it to the data source for preparation. In particular, the driver modifies the escape clauses used to define ODBC-specific SQL. For the driver, an *hstmt* is similar to a statement identifier in embedded SQL code. If the data source supports statement identifiers, the driver can send a statement identifier and parameter values to the data source.

Once a statement is prepared, the application uses *hstmt* to refer to the statement in later function calls. The prepared statement associated with the *hstmt* might be re-executed by calling **SQLExecute** until the application frees the *hstmt* with a call to **SQLFreeStmt** with the SQL_DROP option or until the *hstmt* is used in a call to **SQLPrepare**, **SQLExecDirect**, or one of the catalog functions (**SQLColumns**, **SQLTables**, and so on). Once the application prepares a statement, it can request information about the format of the result set.

Some drivers cannot return syntax errors or access violations when the application calls **SQLPrepare**. A driver might handle syntax errors and access violations, only syntax errors, or neither syntax errors nor access violations. Therefore, an application must be able to handle these conditions when it calls subsequent related functions such as **SQLNumResultCols**, **SQLDescribeCol**, **SQLColAttributes**, and **SQLExecute**.

Depending on the capabilities of the driver and data source and on whether the application has called **SQLBindParameter**, parameter information (such as data types) might be checked when the statement is prepared or when it is executed. For maximum interoperability, an application should unbind all parameters that applied to an old SQL statement before it prepares a new SQL statement on the same *hstmt*. This action prevents errors that are caused by old parameter information being applied to the new statement.

Warning: *Committing or rolling back a transaction, either by calling **SQLTransact** or by using the SQL_AUTOCOMMIT connection option, can cause the data source to delete the access plans for all statement handles on a connection handle. For more information, see "SQL_CURSOR_COMMIT_BEHAVIOR" and "SQL_CURSOR_ROLLBACK_BEHAVIOR" on page 12-196.*

Code Example

See **SQLBindParameter** and **SQLPutData**.

Related Functions

For Information About	See
Allocating a statement handle	**SQLAllocStmt**
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Returning the number of rows that a statement affects	**SQLRowCount**
Setting a cursor name	**SQLSetCursorName**
Assigning storage for a parameter	**SQLBindParameter**
Executing a commit or rollback operation	**SQLTransact**

♦

Level 2

SQLPrimaryKeys

SQLPrimaryKeys returns the column names that comprise the primary key for a table. The driver returns the information as a result set. This function does not support returning primary keys from multiple tables in a single call.

Syntax

```
RETCODE SQLPrimaryKeys(hstmt, szTableQualifier,
cbTableQualifier, szTableOwner, cbTableOwner, szTableName,
cbTableName)
```

The **SQLPrimaryKeys** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szTableQualifier*	Input	Qualifier name. If a driver supports qualifiers for some tables but not for others, as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbTableQualifier*	Input	Length of *szTableQualifier.*
UCHAR FAR *	*szTableOwner*	Input	Table owner. If a driver supports owners for some tables but not for others, as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have owners.
SWORD	*cbTableOwner*	Input	Length of *szTableOwner.*
UCHAR FAR *	*szTableName*	Input	Table name.
SWORD	*cbTableName*	Input	Length of *szTableName*.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.

(1 of 2)

SQLSTATE	Error	Description
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS. The value of one of the name-length arguments exceeded the maximum-length value for the corresponding qualifier or name.
S1C00	Driver not capable	A table qualifier was specified, but the driver or data source does not support qualifiers. A table owner was specified, but the driver or data source does not support owners. The driver or data source did not support the combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options.
S1T00	Time-out expired	The time-out period expired before the data source returned the requested result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

SQLPrimaryKeys returns the results as a standard result set, ordered by TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, and KEY_SEQ. The following table lists the columns in the result set.

The lengths of VARCHAR columns that the table shows are maximums; the actual lengths depend on the data source. To determine the actual lengths of the TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, and COLUMN_NAME columns, call **SQLGetInfo** with the SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

Column Name	Data Type	Comments
TABLE_QUALIFIER	VARCHAR(128)	Primary-key table-qualifier identifier; null if not applicable to the data source.
TABLE_OWNER	VARCHAR(128)	Primary-key table-owner identifier; null if not applicable to the data source.
TABLE_NAME	VARCHAR(128) not NULL	Primary-key table identifier.
COLUMN_NAME	VARCHAR(128) not NULL	Primary-key column identifier.
KEY_SEQ	SMALLINT not NULL	Column sequence number in key (starting with 1).
PK_NAME	VARCHAR(128)	Primary-key identifier. Null if not applicable to the data source.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning table statistics and indexes	**SQLStatistics**

♦

Level 2

SQLProcedureColumns

SQLProcedureColumns returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures. The driver returns the information as a result set on the specified *hstmt*.

Syntax

```
RETCODE SQLProcedureColumns(hstmt, szProcQualifier,
cbProcQualifier, szProcOwner, cbProcOwner, szProcName,
cbProcName, szColumnName, cbColumnName)
```

The **SQLProcedureColumns** function accepts the following arguments:

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szProcQualifier*	Input	Procedure qualifier name. If a driver supports qualifiers for some procedures but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those procedures that do not have qualifiers.
SWORD	*cbProcQualifier*	Input	Length of *szProcQualifier.*
UCHAR FAR *	*szProcOwner*	Input	String search pattern for procedure owner names. If a driver supports owners for some procedures but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those procedures that do not have owners.
SWORD	*cbProcOwner*	Input	Length of *szProcOwner.*
UCHAR FAR *	*szProcName*	Input	String search pattern for procedure names.
SWORD	*cbProcName*	Input	Length of *szProcName*.
UCHAR FAR *	*szColumnName*	Input	String search pattern for column names.
SWORD	*cbColumnName*	Input	Length of *szColumnName*.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
24000	Invalid cursor state	(DM) A cursor was open on the *hstmt* and **SQLFetch** or **SQLExtendedFetch** had been called. A cursor was open on the *hstmt,* but **SQLFetch** or **SQLExtendedFetch** was not called.
IM001	Driver does not support this function	(DM) The driver associated with the *hstmt* does not support the function.
S1000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.

(1 of 2)

SQLSTATE	Error	Description
S1001	Memory-allocation failure	The driver was unable to allocate memory required to support execution or completion of the function.
S1010	Function sequence error	(DM) **SQLExecute**, **SQLExecDirect**, or **SQLSetPos** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value of one of the name length arguments was less than 0, but not equal to SQL_NTS. The value of one of the name length arguments exceeded the maximum length value for the corresponding qualifier or name.
S1C00	Driver not capable	A procedure qualifier was specified and the driver or data source does not support qualifiers. A procedure owner was specified and the driver or data source does not support owners. A string search pattern was specified for the procedure owner, procedure name, or column name and the data source does not support search patterns for one or more of those arguments. The driver or data source did not support the combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Comments

This function is typically used before statement execution to retrieve information about procedure parameters and columns from the data source's catalog.

***Important:** SQLProcedureColumns might not return all columns that a procedure uses. For example, a driver might only return information about the parameters that a procedure uses and not the columns in a result set that it generates.*

The *szProcOwner*, *szProcName*, and *szColumnName* arguments accept search patterns. For more information about valid search patterns, see "Search Pattern Arguments" earlier in this chapter.

SQLProcedureColumns returns the results as a standard result set, ordered by PROCEDURE_QUALIFIER, PROCEDURE_OWNER, PROCEDURE_NAME, and COLUMN_TYPE. The following table lists the columns in the result set. The driver can define additional columns beyond column 13 (REMARKS).

The lengths of VARCHAR columns as the table shows are maximums; the actual lengths depend on the data source. To determine the actual lengths of the PROCEDURE_QUALIFIER, PROCEDURE_OWNER, PROCEDURE_NAME, and COLUMN_NAME columns, an application can call **SQLGetInfo** with the SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, SQL_MAX_PROCEDURE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

Column Name	Data Type	Comments
PROCEDURE_QUALIFIER	VARCHAR(128)	Procedure qualifier identifier; NULL if not applicable to the data source. If a driver supports qualifiers for some procedures but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those procedures that do not have qualifiers.
PROCEDURE_OWNER	VARCHAR(128)	Procedure owner identifier; NULL if not applicable to the data source. If a driver supports owners for some procedures but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those procedures that do not have owners.
PROCEDURE_NAME	VARCHAR(128) not NULL	Procedure identifier.

(1 of 3)

Column Name	Data Type	Comments
COLUMN_NAME	VARCHAR(128) not NULL	Procedure column identifier.
COLUMN_TYPE	SMALLINT not NULL	Defines the procedure column as parameter or a result set column: ■ SQL_PARAM_TYPE_UNKNOWN: The procedure column is a parameter whose type is unknown. (ODBC 1.0) ■ SQL_PARAM_INPUT: The procedure column is an input parameter. (ODBC 1.0) ■ SQL_PARAM_INPUT_OUTPUT: the procedure column is an input/output parameter. (ODBC 1.0) ■ SQL_PARAM_OUTPUT: The procedure column is an output parameter. (ODBC 1.0) ■ SQL_RETURN_VALUE: The procedure column is the return value of the procedure. (ODBC 2.0) ■ SQL_RESULT_COL: The procedure column is a result set column. (ODBC 1.0)
DATA_TYPE	SMALLINT not NULL	SQL data type. This can be an ODBC SQL data type or a driver-specific SQL data type. For a list of valid ODBC SQL data types, see "SQL Data Types" on page B-2. For information about driver-specific SQL data types, see the driver's documentation.
TYPE_NAME	VARCHAR(128) not NULL	Data source–dependent data type name; for example, "CHAR", "VARCHAR", "MONEY", "LONG VARBINARY", or "CHAR ( ) FOR BIT DATA".
PRECISION	INTEGER	The precision of the procedure column on the data source. NULL is returned for data types where precision is not applicable. For more information concerning precision, see "Precision, Scale, Length, and Display Size" on page B-5.
LENGTH	INTEGER	The length in bytes of data transferred on an **SQLGetData** or **SQLFetch** operation if SQL_C_DEFAULT is specified. For numeric data, this size may be different than the size of the data stored on the data source. For more information, see "Precision, Scale, Length, and Display Size" on page B-5.

(2 of 3)

Column Name	Data Type	Comments
SCALE	SMALLINT	The scale of the procedure column on the data source. NULL is returned for data types where scale is not applicable. For more information concerning scale, see "Precision, Scale, Length, and Display Size" on page B-5.
RADIX	SMALLINT	For numeric data types, either 10 or 2. If it is 10, the values in PRECISION and SCALE give the number of decimal digits allowed for the column. For example, a DECIMAL(12,5) column would return a RADIX of 10, a PRECISION of 12, and a SCALE of 5; a FLOAT column could return a RADIX of 10, a PRECISION of 15 and a SCALE of NULL. If it is 2, the values in PRECISION and SCALE give the number of bits allowed in the column. For example, a FLOAT column could return a RADIX of 2, a PRECISION of 53, and a SCALE of NULL. NULL is returned for data types where radix is not applicable.
NULLABLE	SMALLINT not NULL	Whether the procedure column accepts a NULL value: ■ SQL_NO_NULLS: The procedure column does not accept NULL values. ■ SQL_NULLABLE: The procedure column accepts NULL values. ■ SQL_NULLABLE_UNKNOWN: It is not known if the procedure column accepts NULL values.
REMARKS	VARCHAR(254)	A description of the procedure column.

(3 of 3)

Code Example

See **SQLProcedures**.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning a list of procedures in a data source	**SQLProcedures**

♦

Level 2

SQLProcedures

SQLProcedures returns the list of procedure names stored in a specific data source. *Procedure* is a generic term used to describe an *executable object*, or a named entity that can be invoked using input and output parameters, and which can return result sets similar to the results that SELECT statements return.

Syntax

```
RETCODE SQLProcedures(hstmt, szProcQualifier,
cbProcQualifier, szProcOwner, cbProcOwner, szProcName,
cbProcName)
```

The **SQLProcedures** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szProcQualifier*	Input	Procedure qualifier. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbProcQualifier*	Input	Length of *szProcQualifier.*
UCHAR FAR *	*szProcOwner*	Input	String search pattern for procedure-owner names. If a driver supports owners for some procedures but not for others, as when the driver retrieves data from different DBMSs, an empty string ("") denotes those procedures that do not have owners.
SWORD	*cbProcOwner*	Input	Length of *szProcOwner.*
UCHAR FAR *	*szProcName*	Input	String search pattern for procedure names.
SWORD	*cbProcName*	Input	Length of *szProcName.*

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver was unable to allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

(1 of 2)

SQLSTATE	Error	Description
S1090	Invalid string or buffer length	(DM) The value of one of the name-length arguments was less than 0 but not equal to SQL_NTS. The value of one of the name-length arguments exceeded the maximum-length value for the corresponding qualifier or name.
S1C00	Driver not capable	A procedure qualifier was specified, but the driver or data source does not support qualifiers. A procedure owner was specified, but the driver or data source does not support owners. A string search pattern was specified for the procedure owner or procedure name, but the data source does not support search patterns for one or more of those arguments. The driver or data source did not support the combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options.
S1T00	Time-out expired	The time-out period expired before the data source returned the requested result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

SQLProcedures lists all procedures in the requested range. A user might or might not have permission to execute any of these procedures. To check accessibility, an application can call **SQLGetInfo** and check the SQL_ACCESSIBLE_PROCEDURES information value. Otherwise, the application must be able to handle a situation in which the user selects a procedure that it cannot execute.

Important: ***SQLProcedures*** *might not return all procedures. Applications can use any valid procedure, regardless of whether* ***SQLProcedures*** *returns it.*

SQLProcedures returns the results as a standard result set, ordered by PROCEDURE_QUALIFIER, PROCEDURE_OWNER, and PROCEDURE_NAME. The following table lists the columns in the result set.

The lengths of VARCHAR columns shown in the table are maximums; the actual lengths depend on the data source. To determine the actual lengths of the PROCEDURE_QUALIFIER, PROCEDURE_OWNER, and PROCEDURE_NAME columns, an application can call **SQLGetInfo** with the SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, and SQL_MAX_PROCEDURE_NAME_LEN options.

Column Name	Data Type	Comments
PROCEDURE_QUALIFIER	VARCHAR(128)	Procedure qualifier identifier; NULL if not applicable to the data source. If a driver supports qualifiers for some procedures but not for others, as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those procedures that do not have qualifiers.
PROCEDURE_OWNER	VARCHAR(128)	Procedure owner identifier; NULL if not applicable to the data source. If a driver supports owners for some procedures but not for others, as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those procedures that do not have owners.
PROCEDURE_NAME	VARCHAR(128) not NULL	Procedure identifier.
NUM_INPUT_PARAMS	N/A	Reserved for future use. Applications should not rely on the data returned in these result columns.
NUM_OUTPUT_PARAMS	N/A	Reserved for future use. Applications should not rely on the data returned in these result columns.

(1 of 2)

Column Name	Data Type	Comments
NUM_RESULT_SETS	N / A	Reserved for future use. Applications should not rely on the data returned in these result columns.
REMARKS	VARCHAR(254)	A description of the procedure.
PROCEDURE_TYPE	SMALLINT	Defines the procedure type: ■ SQL_PT_UNKNOWN: It cannot be determined whether the procedure returns a value. ■ SQL_PT_PROCEDUR2E: The returned object is a procedure; that is, it does not have a return value. ■ SQL_PT_FUNCTION: The returned object is a function; that is, it has a return value.

(2 of 2)

The *szProcOwner* and *szProcName* arguments accept search patterns. For more information about valid search patterns, see "Search Pattern Arguments" on page 12-8.

Code Example

In this example, an application uses the procedure **AddEmployee** to insert data into the **EMPLOYEE** table. The procedure contains input parameters for **NAME**, **AGE**, and **BIRTHDAY** columns. It also contains one output parameter that returns a remark about the new employee. The example also shows the use of a return value from a stored procedure. For the return value and each parameter in the procedure, the application calls **SQLBindParameter** to specify the ODBC C data type and the SQL data type of the parameter and to specify the storage location and length of the parameter. The application assigns data values to the storage locations for each parameter and calls **SQLExecDirect** to execute the procedure. If **SQLExecDirect** returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the return value and the value of each output or input/output parameter is automatically put into the storage location defined for the parameter in **SQLBindParameter**.

```
#define NAME_LEN 30
#define REM_LEN 128

UCHAR       szName[NAME_LEN], szRemark[REM_LEN];
SWORD       sAge, sEmpId;
SDWORD      cbEmpId, cbName, cbAge = 0, cbBirthday = 0, cbRemark;
DATE_STRUCT dsBirthday;
/* Define parameter for return value (Employee ID) from procedure. */

SQLBindParameter(hstmt, 1, SQL_PARAM_OUTPUT, SQL_C_SLONG, SQL_INTEGER,
                 0, 0, &sEmpId, 0, &cbEmpId);

/* Define data types and storage locations for Name, Age, Birthday */
/* input parameter data.                                           */

SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                 NAME_LEN, 0, szName, 0, &cbName);
SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_SSHORT, SQL_SMALLINT,
                 0, 0, &sAge, 0, &cbAge);
SQLBindParameter(hstmt, 4, SQL_PARAM_INPUT, SQL_C_DATE, SQL_DATE,
                 0, 0, &dsBirthday, 0, &cbBirthday);

/* Define data types and storage location for Remark output parameter */

SQLBindParameter(hstmt, 5, SQL_PARAM_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                 REM_LEN, 0, szRemark, REM_LEN, &cbRemark);

strcpy(szName, "Smith, John D.");    /* Specify first row of */
sAge = 40;                           /* parameter data.      */
dsBirthday.year = 1952;
dsBirthday.month = 2;
dsBirthday.day = 29;
cbName = SQL_NTS;

/* Execute procedure with first row of data. After the procedure */
/* is executed, sEmpId and szRemark will have the values         */
/* returned by AddEmployee.                                       */

retcode = SQLExecDirect(hstmt, "{?=call AddEmployee(?,?,?,?)}",SQL_NTS);

strcpy(szName, "Jones, Bob K.");     /* Specify second row of */
sAge = 52;                           /* parameter data        */
dsBirthday.year = 1940;
dsBirthday.month = 3;
dsBirthday.day = 31;

/* Execute procedure with second row of data. After the procedure */
/* is executed, sEmpId and szRemark will have the new values      */
/* returned by AddEmployee.                                       */

retcode = SQLExecDirect(hstmt,

                        "{?=call AddEmployee(?,?,?,?)}", SQL_NTS);
```

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning information about a driver or data source	**SQLGetInfo**
Returning the parameters and result set columns of a procedure	**SQLProcedureColumns** (extension)
Syntax for invoking stored procedures	Chapter 5, "Executing SQL Statements"

♦

Level 1

SQLPutData

SQLPutData allows an application to send data for a parameter or column to the driver at statement execution time. This function can send character or binary data values in parts to a column with a character, binary, or data-source-specific data type (for example, parameters of SQL_LONGVARBINARY or SQL_LONGVARCHAR).

Syntax

```
    RETCODE SQLPutData(hstmt, rgbValue, cbValue)
```

The **SQLPutData** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
PTR	*rgbValue*	Input	Pointer to storage for the actual data for the parameter or column. The data must use the C data type specified in the *fCType* argument of **SQLBindParameter** (for parameter data) or **SQLBindCol** (for column data).
SDWORD	*cbValue*	Input	Length of *rgbValue*. Specifies the amount of data sent in a call to **SQLPutData**. The amount of data can vary with each call for a given parameter or column. *cbValue* is ignored unless it is SQL_NTS, SQL_NULL_DATA, or SQL_DEFAULT_PARAM; the C data type specified in **SQLBindParameter** or **SQLBindCol** is SQL_C_CHAR or SQL_C_BINARY; or the C data type is SQL_C_DEFAULT and the default C data type for the specified SQL data type is SQL_C_CHAR or SQL_C_BINARY. For all other types of C data, if *cbValue* is not SQL_NULL_DATA or SQL_DEFAULT_PARAM, the driver assumes that the size of *rgbValue* is the size of the C data type specified with *fCType* and sends the entire data value. For more information, see "Converting Data from C to SQL" on page B-31.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated	The data sent for a character or binary parameter or column in one or more calls to **SQLPutData** exceeded the maximum length of the associated character or binary column. The fractional part of the data sent for a numeric or bit parameter or column was truncated. Time-stamp data sent for a date or time parameter or column was truncated.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
22001	String data right truncation	The SQL_NEED_LONG_DATA_LEN information type in **SQLGetInfo** was “Y” and more data was sent for a long parameter (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long, data-source-specific data type) than was specified with the *pcbValue* argument in **SQLBindParameter**.

(1 of 2)

SQLSTATE	Error	Description
22003	Numeric value out of range	**SQLPutData** was called more than once for a parameter or column, and it was not being used to send character C data to a column with a character, binary, or data-source-specific data type or to send binary C data to a column with a character, binary, or data-source-specific data type. The data sent for a numeric parameter or column caused the whole (as opposed to fractional) part of the number to be truncated when assigned to the associated table column.
22005	Error in assignment	The data sent for a parameter or column was incompatible with the data type of the associated table column.
22008	Datetime-field overflow	The data sent for a date, time, or time-stamp parameter or column was, respectively, an invalid date, time, or time stamp.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver could not allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application. **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. **SQLCancel** was called before data was sent for all data-at-execution parameters or columns.
S1009	Invalid argument value	(DM) The argument *rgbValue* was a null pointer, and the argument *cbValue* was not 0, SQL_DEFAULT_PARAM, or SQL_NULL_DATA.
S1010	Function-sequence error	(DM) The previous function call was not a call to **SQLPutData** or **SQLParamData**. The previous function call was a call to **SQLExecDirect** or **SQLExecute** where the return code was SQL_NEED_DATA.
S1090	Invalid string or buffer length	The argument *rgbValue* was not a null pointer, and the argument *cbValue* was less than 0 but not equal to SQL_NTS or SQL_NULL_DATA.
S1T00	Time-out expired	The time-out period expired before the data source completed processing the parameter value. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

For an explanation of how data-at-execution parameter data passes when a statement executes, see "Passing Parameter Values" on page 12-33. For an explanation of how data-at-execution column data is updated or added, see "SQLSetScrollOptions" on page 12-295.

Important: *An application can use **SQLPutData** to send parts of data when sending character C data to a column with a character, binary, or data source-specific data type or when **SQLPutData** sends binary C data to a column with a character, binary, or data source-specific data type. If **SQLPutData** is called more than once under any other conditions, it returns SQL_ERROR and SQLSTATE 22003 (Numeric value out of range).*

Code Example

In the following example, an application prepares an SQL statement to insert data into the **EMPLOYEE** table. The statement contains parameters for the **NAME**, **ID**, and **PHOTO** columns. For each parameter, the application calls **SQLBindParameter** to specify the C and SQL data types of the parameter. It also specifies that the data for the first and third parameters passes at execution time and that the values 1 and 3 pass for later retrieval by **SQLParamData**. These values identify which parameter is being processed.

The application calls **GetNextID** to get the next available employee ID number. It then calls **SQLExecute** to execute the statement. **SQLExecute** returns SQL_NEED_DATA when it needs data for the first and third parameters. The application calls **SQLParamData** to retrieve the value that it stored with **SQLBindParameter**; it uses this value to determine for which parameter to send data. For each parameter, the application calls **InitUserData** to initialize the data routine. It repeatedly calls **GetUserData** and **SQLPutData** to get and send the parameter data. Finally, it calls **SQLParamData** to indicate that it sent all the data for the parameter and to retrieve the value for the next parameter. After data is sent for both parameters, **SQLParamData** returns SQL_SUCCESS.

For the first parameter, **InitUserData** does nothing, and **GetUserData** calls a routine to prompt the user for the employee name. For the third parameter, **InitUserData** calls a routine to prompt the user for the name of a file containing a bitmap photo of the employee and opens the file. **GetUserData** retrieves the next MAX_DATA_LEN bytes of photo data from the file. After it retrieves all the photo data, it closes the photo file.

Some application routines are omitted for clarity.

```
#define MAX_DATA_LEN 1024
SDWORD  cbNameParam, cbID = 0; cbPhotoParam, cbData;
SWORD   sID;
PTR     pToken, InitValue;
UCHAR   Data[MAX_DATA_LEN];

retcode = SQLPrepare(hstmt,
          "INSERT INTO EMPLOYEE (NAME, ID, PHOTO) VALUES (?, ?, ?)",
          SQL_NTS);
if (retcode = = SQL_SUCCESS) {

    /* Bind the parameters. For parameters 1 and 3, pass the       */
    /* parameter number in rgbValue instead of a buffer address.   */

    SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                NAME_LEN, 0, 1, 0, &cbNameParam);
    SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_SSHORT,
                     SQL_SMALLINT, 0, 0, &sID, 0, &cbID);
    SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT,
                     SQL_C_BINARY, SQL_LONGVARBINARY,
                     0, 0, 3, 0, &cbPhotoParam);

    /* Set values so data for parameters 1 and 3 will be passed  */
    /* at execution. Note that the length parameter in the macro */
    /* SQL_LEN_DATA_AT_EXEC is 0. This assumes that the driver   */
    /* returns "N" for the SQL_NEED_LONG_DATA_LEN information    */
    /* type in SQLGetInfo.                                       */

    cbNameParam = cbPhotoParam = SQL_LEN_DATA_AT_EXEC(0);

    sID = GetNextID();  /* Get next available employee ID number. */

    retcode = SQLExecute(hstmt);

    /* For data-at-execution parameters, call SQLParamData to get the */
    /* parameter number set by SQLBindParameter. Call InitUserData.   */
    /* Call GetUserData and SQLPutData repeatedly to get and put all  */
    /* data for the parameter. Call SQLParamData to finish processing */
    /* this parameter and start processing the next parameter.        */

    while (retcode = = SQL_NEED_DATA) {
        retcode = SQLParamData(hstmt, &pToken);
        if (retcode = = SQL_NEED_DATA) {
            InitUserData((SWORD)pToken, InitValue);
            while (GetUserData(InitValue, (SWORD)pToken, Data, &cbData))
```

```
                SQLPutData(hstmt, Data, cbData);
        }
    }
}

VOID InitUserData(sParam, InitValue)
SWORD sParam;
PTR   InitValue;
{
UCHAR  szPhotoFile[MAX_FILE_NAME_LEN];
switch sParam {
    case 3:

        /* Prompt user for bitmap file containing employee photo.    */
        /* OpenPhotoFile opens the file and returns the file handle. */

        PromptPhotoFileName(szPhotoFile);
        OpenPhotoFile(szPhotoFile, (FILE *)InitValue);
        break;
}
}

BOOL GetUserData(InitValue, sParam, Data, cbData)
PTR    InitValue;
SWORD  sParam;
UCHAR  *Data;
SDWORD *cbData;

{
switch sParam {
    case 1:
        /* Prompt user for employee name. */

        PromptEmployeeName(Data);
        *cbData = SQL_NTS;
        return (TRUE);

    case 3:
        /* GetNextPhotoData returns the next piece of photo data and  */
        /* the number of bytes of data returned (up to MAX_DATA_LEN). */

        Done = GetNextPhotoData((FILE *)InitValue, Data,

                                 MAX_DATA_LEN, &cbData);
        if (Done) {
            ClosePhotoFile((FILE *)InitValue);
            return (TRUE);
        }
        return (FALSE);
}
return (FALSE);
}
```

Related Functions

For Information About	See
Canceling statement processing	**SQLCancel**
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Returning the next parameter to send data for	**SQLParamData**
Assigning storage for a parameter	**SQLBindParameter**

♦

Core

SQLRowCount

SQLRowCount returns the number of rows that an UPDATE, INSERT, or DELETE statement affects.

Syntax

```
RETCODE SQLRowCount(hstmt, pcrow)
```

The **SQLRowCount** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
SDWORD FAR *	*pcrow*	Output	For UPDATE, INSERT, and DELETE statements, *pcrow* is the number of rows affected by the request or `-1` if the number of affected rows is not available. For other statements and functions, the driver can define the value of *pcrow*. For example, some data sources might be able to return the number of rows that a SELECT statement or a catalog function returns before fetching the rows. Many data sources cannot return the number of rows in a result set before fetching them; for maximum interoperability, applications should not rely on this behavior.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) The function was called prior to calling **SQLExecute** or **SQLExecDirect** for the *hstmt*. (DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

Usage

If the last executed statement associated with *hstmt* was not an UPDATE, INSERT, or DELETE statement, the value of *pcrow* is driver defined.

Related Functions

For Information About	See
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**

♦

Level 1

SQLSetConnectOption

SQLSetConnectOption sets options that govern aspects of connections.

Syntax

```
RETCODE SQLSetConnectOption(hdbc, fOption, vParam)
```

The **SQLSetConnectOption** function accepts the following arguments.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.
UWORD	*fOption*	Input	Option to set, listed in "Usage."
UDWORD	*vParam*	Input	Value associated with *fOption*. Depending on the value of *fOption*, *vParam* will be a 32-bit integer value or point to a null-terminated character string.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

The driver can return SQL_SUCCESS_WITH_INFO to provide information about the result of setting an option. For example, setting SQL_ACCESS_MODE to read only during a transaction might cause the transaction to be committed. The driver could use SQL_SUCCESS_WITH_INFO, and information returned with **SQLError**, to inform the application of the commit action.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed	The driver did not support the specified value of the *vParam* argument and substituted a similar value (function returns SQL_SUCCESS_WITH_INFO).
08002	Connection in use	The argument *fOption* was SQL_ODBC_CURSORS, and the driver was already connected to the data source.
08003	Connection not open	An *fOption* value was specified that required an open connection, but the *hdbc* was not in a connected state.
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
IM009	Unable to load translation shared library	The driver was unable to load the translation shared library that was specified for the connection. This error can only be returned when *fOption* is SQL_TRANSLATE_DLL.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.

(1 of 2)

SQLSTATE	Error	Description
S1009	Invalid argument value	Given the specified *fOption* value, an invalid value was specified for the argument *vParam*. (INFORMIX-CLI returns this SQLSTATE only for connection and statement options that accept a discrete set of values, such as SQL_ACCESS_MODE. For all other connection and statement options, the driver must verify the value of the argument *vParam*.)
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for an *hstmt* associated with the *hdbc* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) **SQLBrowseConnect** was called for the *hdbc* and returned SQL_NEED_DATA. This function was called before **SQLBrowseConnect** returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.
S1011	Operation invalid at this time	The argument *fOption* was SQL_TXN_ISOLATION, and a transaction was open.
S1092	Option type out of range	(DM) The value specified for the argument *fOption* was in the block of numbers reserved for ODBC connection and statement options, but was not valid for the version of ODBC that the driver supports.
S1C00	Driver not capable	The driver or data source does not support the value specified for the argument *f*Option.

(2 of 2)

When *fOption* is a statement option, **SQLSetConnectOption** can return any SQLSTATE that **SQLSetStmtOption** returns.

Usage

The following table shows the currently defined options. ODBC reserves options from 0 to 999.

An application can call **SQLSetConnectOption** and include a statement option. The driver sets the statement option for any statement handles associated with the specified connection handle and establishes the statement option as a default for any statement handles later allocated for that connection handle. For a list of statement options, see "Usage" on page 12-298.

All connection and statement options that the application successfully sets for the connection handle persist until **SQLFreeConnect** is called for the connection handle. For example, if an application calls **SQLSetConnectOption** before it connects to a data source, the option persists even if **SQLSetConnectOption** fails in the driver when the application connects to the data source; if an application sets a driver-specific option, the option persists even if the application connects to a different driver on the connection handle.

Some connection and statement options support substituting a similar value if the data source does not support the specified value of *vParam*. In such cases, the driver returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed). For example, if *fOption* is SQL_PACKET_SIZE and *vParam* exceeds the maximum packet size, the driver substitutes the maximum size. To determine the substituted value, an application calls **SQLGetConnectOption** (for connection options) or **SQLGetStmtOption** (for statement options).

The format of information set through *vParam* depends on the specified *fOption*. **SQLSetConnectOption** accepts option information in one of two formats: a null-terminated character string or a 32-bit integer value. The format of each *fOption* is noted in the following table. Character strings pointed to by the *vParam* argument of **SQLSetConnectOption** have a maximum length of SQL_MAX_OPTION_STRING_LENGTH bytes (excluding the null-termination byte).

In addition to the options described in the following table, INFORMIX-CLI supports an alternative isolation level 1, called cursor stability. To use this isolation level, your INFORMIX-CLI application should call **SQLSetConnectOption** with the *fOption* value set to `1040` and *vParam* set to `1`.

fOption	vParam Contents
SQL_ACCESS_MODE	A 32-bit integer value. SQL_MODE_READ_ONLY is used by the driver or data source as an indicator that the connection is not required to support SQL statements that cause updates to occur. This mode can be used to optimize locking strategies, transaction management, or other areas as appropriate to the driver or data source. The driver is not required to prevent such statements from being submitted to the data source. The behavior of the driver and data source when asked to process SQL statements that are not read-only during a read-only connection is implementation defined. SQL_MODE_READ_WRITE is the default.
SQL_AUTOCOMMIT	A 32-bit integer value that specifies whether to use auto-commit or manual-commit mode: ■ SQL_AUTOCOMMIT_OFF = The driver uses manual-commit mode, and the application must explicitly commit or roll back transactions with **SQLTransact**. ■ SQL_AUTOCOMMIT_ON = The driver uses autocommit mode. Each statement is committed immediately after it is executed. This is the default. Changing from manual-commit mode to autocommit mode commits any open transactions on the connection. ***Important**: Some data sources delete the access plans and close the cursors for all statement handles on a connection handle each time a statement is committed; auto-commit mode can cause this to happen after each statement is executed. For more information, see "SQL_CURSOR_COMMIT_BEHAVIOR" and "SQL_CURSOR_ROLLBACK_BEHAVIOR" on page 12-196.*
SQL_CURRENT_QUALIFIER	A null-terminated character string containing the name of the qualifier to be used by the data source. For a single-tier driver, the qualifier might be a directory, in which case the driver changes its current directory to the directory specified in *vParam*.

(1 of 4)

fOption	**vParam Contents**
SQL_LOGIN_TIMEOUT	A 32-bit integer value corresponding to the number of seconds to wait for a login request to complete before returning to the application. The default is driver dependent and must be nonzero. If *vParam* is 0, the time-out is disabled, and a connection attempt will wait indefinitely. If the specified time-out exceeds the maximum login time-out in the data source, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed).
SQL_ODBC_CURSORS	A 32-bit option specifying how the driver manager uses the ODBC cursor library: ■ SQL_CUR_USE_IF_NEEDED = INFORMIX-CLI uses the ODBC cursor library only if it is needed. If INFORMIX-CLI supports the SQL_FETCH_PRIOR option in **SQLExtendedFetch**, it uses its scrolling capabilities. Otherwise, it uses the cursor library. ■ SQL_CUR_USE_ODBC = INFORMIX-CLI uses the cursor library. ■ SQL_CUR_USE_DRIVER = INFORMIX-CLI uses its scrolling capabilities. This is the default setting. Informix recommends that you use SQL_CUR_USE_DRIVER. For more information about the ODBC cursor library, see the *Microsoft ODBC Programmer's Reference and SDK Guide,* Version 2.0.
SQL_OPT_TRACE	A 32-bit integer value telling INFORMIX-CLI whether to perform tracing: ■ SQL_OPT_TRACE_OFF = Tracing off (the default) ■ SQL_OPT_TRACE_ON = Tracing on When tracing is on, INFORMIX-CLI writes each INFORMIX-CLI function call to the trace file. When tracing is on, INFORMIX-CLI can return SQLSTATE IM013 (Trace-file error) from any function. An application specifies a trace file with the SQL_OPT_TRACEFILE option. If the file already exists, INFORMIX-CLI appends to the file. Otherwise, it creates the file. If tracing is on but no trace file is specified, INFORMIX-CLI writes to the file **sql.log** in the current directory.
SQL_OPT_TRACEFILE	A null-terminated character string containing the name of the trace file.

(2 of 4)

fOption	vParam Contents
SQL_PACKET_SIZE	A 32-bit integer value specifying the network packet size in bytes. Many data sources either do not support this option or can only return the network packet size. If the specified size exceeds the maximum packet size or is smaller than the minimum packet size, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed).
SQL_QUIET_MODE	A 32-bit window handle (*hwnd*). If the window handle is a null pointer, the driver does not display any dialog boxes. If the window handle is not a null pointer, it should be the parent window handle of the application. The driver uses this handle to display dialog boxes. This is the default. If the application has not specified a parent window handle for this option, the driver uses a null parent window handle to display dialog boxes or return in **SQLGetConnectOption**. The SQL_QUIET_MODE connection option does not apply to dialog boxes that **SQLDriverConnect** displays.
SQL_TRANSLATE_OPTION	A 32-bit flag value that is passed to the translation shared library. This option can only be specified if the driver has connected to the data source.

(3 of 4)

fOption	vParam Contents
SQL_TXN_ISOLATION	A 32-bit mask that sets the transaction isolation level for the current *hdbc*. An application must call **SQLTransact** to commit or roll back all open transactions on an *hdbc* before calling **SQLSetConnectOption** with this option. The valid values for *vParam* can be determined by calling **SQLGetInfo** with *fInfotype* equal to SQL_TXN_ISOLATION_OPTIONS. The following terms are used to define transaction isolation levels: ■ *Dirty Read*: Transaction 1 changes a row. Transaction 2 reads the changed row before transaction 1 commits this change. Transaction 1 rolls back the change, and transaction 2 reads a row that is considered to have never existed. ■ *Nonrepeatable Read*: Transaction 1 reads a row. Transaction 2 updates or deletes that row and commits this change. Transaction 1 attempts to reread the row, and it receives different row values or discovers that the row has been deleted. ■ *Phantom*: Transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 inserts a row that matches the search criteria. Transaction 1 re-executes the statement that read the rows, and it receives a different set of rows. *vParam* must be one of the following values: ■ SQL_TXN_READ_UNCOMMITTED = Dirty Reads, Nonrepeatable Reads, and Phantoms are possible. ■ SQL_TXN_READ_COMMITTED = Dirty Reads are not possible. Nonrepeatable Reads and Phantoms are possible. ■ SQL_TXN_REPEATABLE_READ = Dirty Reads and Nonrepeatable Reads are not possible. Phantoms are possible. ■ SQL_TXN_SERIALIZABLE = Transactions are serializable. Dirty Reads, Nonrepeatable Reads, and Phantoms are not possible. ■ SQL_TXN_VERSIONING = Transactions can be serialized, but higher concurrency is possible than with SQL_TXN_SERIALIZABLE. Dirty Reads are not possible. Typically, SQL_TXN_SERIALIZABLE is implemented by using locking protocols that reduce concurrency, and SQL_TXN_VERSIONING is implemented by using non-locking protocol such as record versioning.

(4 of 4)

Data Translation

Data translation is performed for all data moving between the driver and the data source.

The translation option (set with the SQL_TRANSLATE_OPTION option) can be any 32-bit value. Its meaning depends on the translation shared library that you use. A new option can be set at any time and will be applied to the next exchange of data following a call to **SQLSetConnectOption**. A default translation shared library can be specified for the data source in its data-source specification. The driver loads the default translation shared library at connection time. A translation option (SQL_TRANSLATE_OPTION) can also be specified in the data-source specification.

To change the translation shared library for a connection, an application calls **SQLSetConnectOption** with the SQL_TRANSLATE_DLL option after it connects to the data source. The driver attempts to load the specified shared library and, if the attempt fails, the driver returns SQL_ERROR with the SQLSTATE IM009 (Unable to load translation shared library).

If no translation shared library is specified in the ODBC initialization file or by calling **SQLSetConnectOption**, the driver does not attempt to translate data. Any value set for the translation option is ignored.

For more information about translating data, see Chapter 13, “Setup Shared Library Function Reference.”

Code Example

See **SQLConnect**.

Related Functions

For Information About	See
Returning the setting of a connection option	**SQLGetConnectOption**
Returning the setting of a statement option	**SQLGetStmtOption**
Setting a statement option	**SQLSetStmtOption**

♦

Core

SQLSetCursorName

SQLSetCursorName associates a cursor name with an active *hstmt*. If an application does not call **SQLSetCursorName**, the driver generates cursor names as needed for SQL statement processing.

Syntax

```
RETCODE SQLSetCursorName(hstmt, szCursor, cbCursor)
```

The **SQLSetCursorName** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle
UCHAR FAR *	*szCursor*	Input	Cursor name
SWORD	*cbCursor*	Input	Length of *szCursor*

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state	The statement corresponding to *hstmt* was already in an executed or cursor-positioned state.
34000	Invalid cursor name	The cursor name that the argument *szCursor* specified was invalid. For example, the cursor name exceeded the maximum length that the driver defines.
3C000	Duplicate cursor name	The cursor name that the argument *szCursor* specifies already exists.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.

(1 of 2)

SQLSTATE	Error	Description
S1009	Invalid argument value	(DM) The argument *szCursor* was a null pointer.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The argument *cbCursor* was less than 0, but not equal to SQL_NTS.

(2 of 2)

Usage

The only ODBC SQL statements that use a cursor name are a positioned UPDATE and DELETE (for example, UPDATE *table-name*...WHERE CURRENT OF *cursor-name*). If the application does not call **SQLSetCursorName** to define a cursor name, when a SELECT statement executes, the driver generates a name that begins with the letters SQL_CUR and does not exceed 18 characters.

All cursor names within the *hdbc* must be unique. The driver defines the maximum length of a cursor name. For maximum interoperability, cursor names should not exceed 18 characters.

A cursor name remains set until the *hstmt* with which it is associated is dropped, using **SQLFreeStmt** with the SQL_DROP option.

Code Example

In the following example, an application uses **SQLSetCursorName** to set a cursor name for an *hstmt*. It then uses that *hstmt* to retrieve results from the **EMPLOYEE** table. Finally, it performs a positioned update to change the name of 25-year-old John Smith to John D. Smith. The application uses different *hstmts* for the SELECT and UPDATE statements.

```
#define NAME_LEN 30

HSTMT     hstmtSelect,
HSTMT     hstmtUpdate;
UCHAR     szName[NAME_LEN];
SWORD     sAge;
SDWORD    cbName;
SDWORD    cbAge;

/* Allocate the statements and set the cursor name */

SQLAllocStmt(hdbc, &hstmtSelect);
SQLAllocStmt(hdbc, &hstmtUpdate);
SQLSetCursorName(hstmtSelect, "C1", SQL_NTS);

/* SELECT the result set and bind its columns to local storage */

SQLExecDirect(hstmtSelect,
              "SELECT NAME, AGE FROM EMPLOYEE FOR UPDATE",
              SQL_NTS);
SQLBindCol(hstmtSelect, 1, SQL_C_CHAR, szName, NAME_LEN, &cbName);
SQLBindCol(hstmtSelect, 2, SQL_C_SSHORT, &sAge, 0, &cbAge);

/* Read through the result set until the cursor is      */
/* positioned on the row for the 25-year-old John Smith */

do
   retcode = SQLFetch(hstmtSelect);
while ((retcode = = SQL_SUCCESS || retcode = = SQL_SUCCESS_WITH_INFO) &&
       (strcmp(szName, "Smith, John") != 0 || sAge != 25));

/* Perform a positioned update of John Smith's name */

if (retcode = = SQL_SUCCESS || retcode = = SQL_SUCCESS_WITH_INFO) {
    SQLExecDirect(hstmtUpdate,
      "UPDATE EMPLOYEE SET NAME=\"Smith, John D.\" WHERE CURRENT OF C1",
      SQL_NTS);
}
```

Related Functions

For Information About	See
Executing an SQL statement	**SQLExecDirect**
Executing a prepared SQL statement	**SQLExecute**
Returning a cursor name	**SQLGetCursorName**
Setting cursor scrolling options	**SQLSetScrollOptions**

♦

Level 2

SQLSetScrollOptions

In ODBC 2.0, the SQL_CURSOR_TYPE, SQL_CONCURRENCY, SQL_KEYSET_SIZE, and SQL_ROWSET_SIZE options for **SQLSetStmtOption** superseded **SQLSetScrollOptions**. Applications should not call **SQLSetScrollOptions**. ♦

Level 1

SQLSetStmtOption

SQLSetStmtOption sets options that are related to an *hstmt*. To set an option for all the statements associated with a specific *hdbc*, an application can call **SQLSetConnectOption**.

Syntax

```
RETCODE SQLSetStmtOption(hstmt, fOption, vParam)
```

The **SQLSetStmtOption** function accepts the following arguments.

Typedef	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UWORD	*fOption*	Input	Option to set, listed in "Usage."
UDWORD	*vParam*	Input	Value associated with *fOption*. Depending on the value of *fOption, vParam* will be a 32-bit integer value or point to a null-terminated character string.

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed	The driver did not support the specified value of the *vParam* argument and substituted a similar value (function returns SQL_SUCCESS_WITH_INFO).
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	The *fOption* was SQL_CONCURRENCY, SQL_SIMULATE_CURSOR, or SQL_CURSOR_TYPE, and the cursor was open.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.

(1 of 2)

SQLSTATE	Error	Description
S1009	Invalid argument value	Given the specified *fOption* value, an invalid value was specified for the argument *vParam*. (The driver manager returns this SQLSTATE only for statement options that accept a discrete set of values, such as SQL_ASYNC_ENABLE. For all other statement options, the driver must verify the value of the argument *vParam*.)
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1011	Operation invalid at this time	The *fOption* was SQL_CONCURRENCY, SQL_SIMULATE_CURSOR, or SQL_CURSOR_TYPE, and the statement was prepared.
S1092	Option value out of range	(DM) The value specified for the argument *fOption* was in the block of numbers reserved for ODBC connection and statement options, but was not valid for the version of ODBC that the driver supports.
S1C00	Driver not capable	The driver or data source does not support the value specified for the argument *f*Option.

(2 of 2)

Usage

Statement options for an *hstmt* remain in effect until they are changed by another call to **SQLSetStmtOption** or the *hstmt* is dropped by calling **SQLFreeStmt** with the SQL_DROP option. Calling **SQLFreeStmt** with the SQL_CLOSE, SQL_UNBIND, or SQL_RESET_PARAMS options does not reset statement options.

Some statement options support substituting a similar value if the data source does not support the specified value of *vParam*. In such cases, the driver returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed). For example, if *fOption* is SQL_CONCURRENCY, *vParam* is SQL_CONCUR_ROWVER, and the data source does not support this, the driver substitutes SQL_CONCUR_VALUES. To determine the substituted value, an application calls **SQLGetStmtOption**.

The following table shows the currently defined options. ODBC reserves options from 0 to 999.

The format of information set with *vParam* depends on the specified *fOption*. **SQLSetStmtOption** accepts option information in one of two different formats: a null-terminated character string or a 32-bit integer value. The format is noted in the option description. This format applies to the information returned for each option in **SQLGetStmtOption**. Character strings pointed to by the *vParam* argument of **SQLSetStmtOption** have a maximum length of SQL_MAX_OPTION_STRING_LENGTH bytes (excluding the null-termination byte).

fOption	vParam Contents
SQL_BIND_TYPE	A 32-bit integer value that sets the binding orientation to be used when **SQLExtendedFetch** is called on the associated *hstmt*. Column-wise binding is selected by supplying the defined constant SQL_BIND_BY_COLUMN for the argument *vParam*. Row-wise binding is selected by supplying a value for *vParam* specifying the length of a structure or an instance of a buffer into which result columns will be bound. The length specified in *vParam* must include space for all of the bound columns and any padding of the structure or buffer to ensure that when the address of a bound column is incremented with the specified length, the result will point to the beginning of the same column in the next row. When using the **sizeof** operator with structures or unions in ANSI C, this behavior is guaranteed. Column-wise binding is the default binding orientation for **SQLExtendedFetch**.

(1 of 4)

fOption	**vParam Contents**
SQL_CONCURRENCY	A 32-bit integer value that specifies the cursor concurrency: ■ SQL_CONCUR_READ_ONLY = Cursor is read only. No updates are allowed. ■ SQL_CONCUR_LOCK = Cursor uses the lowest level of locking sufficient to ensure that the row can be updated. ■ SQL_CONCUR_ROWVER = Cursor uses optimistic concurrency control, comparing row versions. ■ SQL_CONCUR_VALUES = Cursor uses optimistic concurrency control, comparing values. The default value is SQL_CONCUR_READ_ONLY. This option can also be set through the *fConcurrency* argument in **SQLSetScrollOptions**. This option cannot be specified for an open cursor. If the SQL_CURSOR_TYPE *fOption* is changed to a cursor type that does not support the current value of SQL_CONCURRENCY, the value of SQL_CONCURRENCY is not automatically changed to a supported value, and no error will be reported until **SQLExecDirect** or **SQLPrepare** is called. If the driver supports the SELECT_FOR_UPDATE statement, and such a statement is executed while the value of SQL_CONCURRENCY is set to SQL_CONCUR_READ_ONLY, an error will be returned. If the value of SQL_CONCURRENCY is changed to a value that the driver supports for some value of SQL_CURSOR_TYPE, but not for the current value of SQL_CURSOR_TYPE, the value of SQL_CURSOR_TYPE is not automatically changed to a supported value, and no error will be reported until **SQLExecDirect** or **SQLPrepare** is called. If the data source does not support the specified concurrency, the driver substitutes a different concurrency and returns SQLSTATE 01S02 (Option value changed). For SQL_CONCUR_VALUES, the driver substitutes SQL_CONCUR_ROWVER, and vice versa. For SQL_CONCUR_LOCK, the driver substitutes, in order, SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES.

(2 of 4)

fOption	**vParam Contents**
SQL_CURSOR_TYPE	A 32-bit integer value that specifies the cursor type: ■ SQL_CURSOR_FORWARD_ONLY = The cursor only scrolls forward. ■ SQL_CURSOR_STATIC = The data in the result set is static. ■ SQL_CURSOR_KEYSET_DRIVEN = The driver saves and uses the keys for the number of rows specified in the SQL_KEYSET_SIZE statement option. ■ SQL_CURSOR_DYNAMIC = The driver only saves and uses the keys for the rows in the row set. The default value is SQL_CURSOR_FORWARD_ONLY. This option cannot be specified for an open cursor and can also be set through the *crowKeyset* argument in **SQLSetScrollOptions**. If the data source does not support the specified cursor type, the driver substitutes a different cursor type and returns SQLSTATE 01S02 (Option value changed). For a mixed or dynamic cursor, the driver substitutes a key-set-driven cursor. A key-set-driven cursor is always set to zero.
SQL_KEYSET_SIZE UNSUPPORTED	A 32-bit integer value that specifies the number of rows in the key set for a key-set-driven cursor. The key-set size is 0 and cannot be changed. If the specified size exceeds the maximum key-set size, the driver substitutes that size and returns SQLSTATE 01S02 (Option value changed). **SQLExtendedFetch** returns an error if the key-set size is greater than 0 and less than the row-set size.
SQL_MAX_LENGTH	A 32-bit integer value that specifies the maximum amount of data that the driver returns from a character or binary column. If *vParam* is less than the length of the available data, **SQLFetch** or **SQLGetData** truncates the data and returns SQL_SUCCESS. If *vParam* is 0 (the default), the driver attempts to return all available data. If the specified length is less than the minimum amount of data that the data source can return (the minimum is 254 bytes on many data sources), or greater than the maximum amount of data that the data source can return, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed). This option is intended to reduce network traffic and should only be supported when the data source (as opposed to the driver) in a multiple-tier driver can implement it. To truncate data, an application should specify the maximum buffer length in the *cbValueMax* argument in **SQLBindCol** or **SQLGetData**. In ODBC 1.0, this statement option only applied to SQL_LONGVARCHAR and SQL_LONGVARBINARY columns.

(3 of 4)

fOption	vParam Contents
SQL_MAX_ROWS	A 32-bit integer value corresponding to the maximum number of rows to return to the application for a SELECT statement. If *vParam* equals 0 (the default), then the driver returns all rows. This option is intended to reduce network traffic. Conceptually, it is applied when the result set is created and limits the result set to the first *vParam* rows. If the specified number of rows exceeds the number of rows that the data source can return, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed).
SQL_NOSCAN	A 32-bit integer value that specifies whether the driver does not scan SQL strings for escape clauses: ■ SQL_NOSCAN_OFF = The driver scans SQL strings for escape clauses (the default). ■ SQL_NOSCAN_ON = The driver does not scan SQL strings for escape clauses. Instead, the driver sends the statement directly to the data source.
SQL_RETRIEVE_DATA	A 32-bit integer value: ■ SQL_RD_ON = **SQLExtendedFetch** retrieves data after it positions the cursor to the specified location. This is the default. ■ SQL_RD_OFF = **SQLExtendedFetch** does not retrieve data after it positions the cursor. By setting SQL_RETRIEVE_DATA to SQL_RD_OFF, an application can verify if a row exists without incurring the overhead of retrieving rows.
SQL_ROWSET_SIZE	A 32-bit integer value that specifies the number of rows in the row set. This is the number of rows returned by each call to **SQLExtendedFetch**. The default value is 1. If the specified row set size exceeds the maximum row set size that the data source supports, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed). This option can be specified for an open cursor and can also be set through the *crowRowset* argument in **SQLSetScrollOptions**.

(4 of 4)

Code Example

See **SQLExtendedFetch**.

Related Functions

For Information About	See
Canceling statement processing	**SQLCancel**
Returning the setting of a connection option	**SQLGetConnectOption**
Returning the setting of a statement option	**SQLGetStmtOption**
Setting a connection option	**SQLSetConnectOption**

♦

Level 1

SQLSpecialColumns

SQLSpecialColumns retrieves the following information about columns within a specified table:

- The optimal set of columns that uniquely identifies a row in the table
- Columns that are automatically updated when any value in the row is updated by a transaction

Syntax

```
RETCODE SQLSpecialColumns(hstmt, fColType, szTableQualifier,
cbTableQualifier, szTableOwner, cbTableOwner, szTableName,
cbTableName, fScope, fNullable)
```

The **SQLSpecialColumns** function accepts the following arguments.

Deputed	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle
UWORD	*fColType*	Input	Type of column to return. Must be one of the following values: ■ SQL_BEST_ROWID: Returns the optimal column or set of columns that, by retrieving values from the column or columns, allows any row in the specified table to be uniquely identified. A column can be either a pseudo-column specifically designed for this purpose or the column or columns of any unique index for the table. ■ SQL_ROWVER: Returns the column or columns in the specified table, if any, that are automatically updated by the data source when any value in the row is updated by any transaction.
UCHAR FAR *	*szTableQualifier*	Input	Qualifier name for the table. If a driver supports qualifiers for some tables but not for others, as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbTableQualifier*	Input	Length of *szTableQualifier*

(1 of 2)

Deputed	Argument	Use	Description
UCHAR FAR *	*szTableOwner*	Input	Owner name for the table. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have owners.
SWORD	*cbTableOwner*	Input	Length of *szTableOwner*
UCHAR FAR *	*szTableName*	Input	Table name
SWORD	*cbTableName*	Input	Length of *szTableName*
UWORD	*fScope*	Input	Minimum required scope of the row ID. The returned row ID might be of greater scope. It must be one of the following: ■ SQL_SCOPE_CURROW: The row ID is guaranteed to be valid only while positioned on that row. A later reselect using row ID might not return a row if the row was updated or deleted by another transaction. ■ SQL_SCOPE_TRANSACTION: The row ID is guaranteed to be valid for the duration of the current transaction. ■ SQL_SCOPE_SESSION: The row ID is guaranteed to be valid for the duration of the session (across transaction boundaries).
UWORD	*fNullable*	Input	Determines whether to return special columns that can have a NULL value. It must be one of the following: ■ SQL_NO_NULLS: Exclude special columns that can have NULL values. ■ SQL_NULLABLE: Return special columns even if they can have NULL values.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory- allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1010	Function-sequence error	(DM) **SQLExecute or SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

(1 of 2)

SQLSTATE	Error	Description
S1090	Invalid string or buffer length	(DM) The value of one of the length arguments was less than 0 but not equal to SQL_NTS. The value of one of the length arguments exceeded the maximum-length value for the corresponding qualifier or name. The maximum length of each qualifier or name can be obtained by calling **SQLGetInfo** with the *fInfoType* values: SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, or SQL_MAX_TABLE_NAME_LEN
S1097	Column type out of range	(DM) An invalid *fColType* value was specified.
S1098	Scope type out of range	(DM) An invalid *fScope* value was specified.
S1099	Nullable type out of range	(DM) An invalid *fNullable* value was specified.
S1C00	Driver not capable	A table qualifier was specified, but the driver or data source does not support qualifiers. A table owner was specified, but the driver or data source does not support owners. The driver or data source did not support the combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options.
S1T00	Time-out expired	The time-out period expired before the data source returned the requested result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

SQLSpecialColumns is provided so that applications can provide their own custom scrollable-cursor functionality, similar to the functionality that **SQLExtendedFetch** and **SQLSetStmtOption** provide.

When the *fColType* argument is SQL_BEST_ROWID, **SQLSpecialColumns** returns the column or columns that uniquely identify each row in the table. These columns can always be used in a SELECT list or WHERE clause. However, **SQLColumns** does not necessarily return these columns. If no columns uniquely identify each row in the table, **SQLSpecialColumns** returns a row set with no rows; a subsequent call to **SQLFetch** or **SQLExtendedFetch** on the *hstmt* returns SQL_NO_DATA_FOUND.

If the *fColType*, *fScope*, or *fNullable* arguments specify characteristics that the data source does not support, **SQLSpecialColumns** returns a result set with no rows (as opposed to the function returning SQL_ERROR with SQLSTATE S1C00 (Driver not capable)). A subsequent call to **SQLFetch** or **SQLExtendedFetch** on the *hstmt* returns SQL_NO_DATA_FOUND.

SQLSpecialColumns returns the results as a standard result set, ordered by SCOPE. The following table lists the columns in the result set.

The lengths of VARCHAR columns shown in the table are maximums; the actual lengths depend on the data source. To determine the actual length of the COLUMN_NAME column, an application can call **SQLGetInfo** with the SQL_MAX_COLUMN_NAME_LEN option.

Column Name	Informix SQL Data Type	Comments
SCOPE	SMALLINT	Actual scope of the row ID. Contains one of the following values: SQL_SCOPE_CURROW SQL_SCOPE_TRANSACTION SQL_SCOPE_SESSION NULL is returned when *fColType* is SQL_ROWVER. For a description of each value, see the description of "fScope" on page 12-305.
COLUMN_NAME	VARCHAR(128) not NULL	Column identifier.
DATA_TYPE	SMALLINT not NULL	INFORMIX-CLI SQL data type. See "SQL Data Types" on page B-2.
TYPE_NAME	VARCHAR(128) not NULL	Informix SQL data type. See "SQL Data Types" on page B-2.

(1 of 2)

Column Name	Informix SQL Data Type	Comments
PRECISION	INTEGER	The precision of the column on the data source. NULL is returned for data types where precision is not applicable. For more information concerning precision, see "Precision, Scale, Length, and Display Size" on page B-5."
LENGTH	INTEGER	The length in bytes of data transferred on an **SQLGetData** or **SQLFetch** operation if SQL_C_DEFAULT is specified. For numeric data, this size might be different than the size of the data stored on the data source. This value is the same as the PRECISION column for character or binary data. For more information, see "Precision, Scale, Length, and Display Size" on page B-5.
SCALE	SMALLINT	The scale of the column on the data source. NULL is returned for data types where scale is not applicable. For more information concerning scale, see "Precision, Scale, Length, and Display Size" on page B-5.
PSEUDO_COLUMN	SMALLINT	Returns one of the following values to indicate whether the column is a pseudo-column: SQL_PC_UNKNOWN SQL_PC_PSEUDO SQL_PC_NOT_PSEUDO **Important:** *For maximum interoperability, pseudo-columns should not be quoted with the identifier quote character that* ***SQLGetInfo*** *returns.*

(2 of 2)

Once the application retrieves values for SQL_BEST_ROWID, the application can use these values to reselect that row within the defined scope. The SELECT statement is guaranteed to return either no rows or one row.

If an application reselects a row based on the row ID column or columns and the row is not found, the application can assume that the row was deleted or the row ID columns were modified. The opposite is not true: Even if the row ID has not changed, the other columns in the row might have changed.

Columns returned for column type SQL_BEST_ROWID are useful for applications that need to scroll forward and backward within a result set to retrieve the most recent data from a set of rows. The row ID column or columns are guaranteed not to change while positioned on that row.

The row ID column or columns might remain valid even when the cursor is not positioned on the row; the application can determine this by checking the SCOPE column in the result set.

Columns returned for column type SQL_ROWVER are useful for applications that need the ability to check if any columns in a given row have been updated while the row was reselected using the row ID. For example, after re-selecting a row using row ID, the application can compare the previous values in the SQL_ROWVER columns to the ones last fetched. If the value in a SQL_ROWVER column differs from the previous value, the application can alert the user that data on the display has changed.

Code Example

For a code example of a similar function, see **SQLColumns**.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning the columns in a table or tables	**SQLColumns**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning the columns of a primary key	**SQLPrimaryKeys**

♦

Level 1

SQLStatistics

SQLStatistics retrieves a list of statistics about a single table and the indexes associated with the table. The driver returns this information as a result set.

Syntax

```
RETCODE SQLStatistics(hstmt, szTableQualifier,
cbTableQualifier,szTableOwner, cbTableOwner, szTableName,
cbTableName, fUnique, fAccuracy)
```

The **SQLStatistics** function accepts the following arguments.

Typedef	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szTableQualifier*	Input	Qualifier name. If a driver supports qualifiers for some tables but not for others, as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbTableQualifier*	Input	Length of *szTableQualifier.*
UCHAR FAR *	*szTableOwner*	Input	Owner name. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have owners.
SWORD	*cbTableOwner*	Input	Length of *szTableOwner.*
UCHAR FAR *	*szTableName*	Input	Table name.
SWORD	*cbTableName*	Input	Length of *szTableName.*

(1 of 2)

Typedef	Argument	Use	Description
UWORD	*fUnique*	Input	Type of index: SQL_INDEX_UNIQUE or SQL_INDEX_ALL
UWORD	*fAccuracy*	Input	The importance of the CARDINALITY and PAGES columns in the result set: ■ SQL_ENSURE requests that the driver unconditionally retrieve the statistics. ■ SQL_QUICK requests that the driver retrieve results only if they are readily available from the server. In this case, the driver does not ensure that the values are current.

(2 of 2)

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory- allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS. The value of one of the name-length arguments exceeded the maximum-length value for the corresponding qualifier or name.
S1100	Uniqueness option type out of range	(DM) An invalid *fUnique* value was specified.

(1 of 2)

SQLSTATE	Error	Description
S1101	Accuracy option type out of range	(DM) An invalid *fAccuracy* value was specified.
S1C00	Driver not capable	A table qualifier was specified, but the driver or data source does not support qualifiers. A table owner was specified, but the driver or data source does not support owners. The driver or data source did not support the combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options.
S1T00	Time-out expired	The time-out period expired before the data source returned the requested result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

SQLStatistics returns information about a single table as a standard result set, ordered by NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME, and SEQ_IN_INDEX. The result set combines statistics information for the table with information about each index. The following table lists the columns in the result set.

Important: *SQLStatistics might not return all indexes. Applications can use any valid index, regardless of whether* ***SQLStatistics*** *returns it.*

The lengths of VARCHAR columns shown in the table are maximums; the actual lengths depend on the data source. To determine the actual lengths of the TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, and COLUMN_NAME columns, an application can call **SQLGetInfo** with the SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

Column Name	Informix SQL Data Type	Comments
TABLE_QUALIFIER	VARCHAR(128)	Table-qualifier identifier of the table to which the statistic or index applies; NULL if not applicable to the data source. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have qualifiers.
TABLE_OWNER	VARCHAR(128)	Table-owner identifier of the table to which the statistic or index applies; NULL if not applicable to the data source. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have owners.
TABLE_NAME	VARCHAR(128) not NULL	Table identifier of the table to which the statistic or index applies
NON_UNIQUE	SMALLINT	Indicates whether the index prohibits duplicate values: ■ TRUE if the index values can be non-unique. ■ FALSE if the index values must be unique. ■ NULL is returned if TYPE is SQL_TABLE_STAT.

(1 of 3)

Column Name	Informix SQL Data Type	Comments
INDEX_QUALIFIER	VARCHAR(128)	The identifier that is used to qualify the index name doing a DROP INDEX; NULL is returned if an index qualifier is not supported by the data source or if TYPE is SQL_TABLE_STAT. If a non-null value is returned in this column, it must be used to qualify the index name on a DROP INDEX statement; otherwise, the TABLE_OWNER name should be used to qualify the index name.
INDEX_NAME	VARCHAR(128)	Index identifier; NULL is returned if TYPE is SQL_TABLE_STAT.
TYPE	SMALLINT not NULL	Type of information being returned: ■ SQL_TABLE_STAT indicates a statistic for the table. ■ SQL_INDEX_CLUSTERED indicates a clustered index. ■ SQL_INDEX_HASHED indicates a hashed index. SQL_INDEX_OTHER indicates another type of index.
SEQ_IN_INDEX	SMALLINT	Column-sequence number in index (starting with 1); NULL is returned if TYPE is SQL_TABLE_STAT.
COLUMN_NAME	VARCHAR(128)	Column identifier. If the column is based on an expression, such as SALARY + BENEFITS, the expression is returned; if the expression cannot be determined, an empty string is returned. If the index is a filtered index, each column in the filter condition is returned; this might require more than one row. NULL is returned if TYPE is SQL_TABLE_STAT.

(2 of 3)

Column Name	Informix SQL Data Type	Comments
COLLATION	CHAR(1)	Sort sequence for the column; "A" for ascending; "D" for descending; NULL is returned if column sort sequence is not supported by the data source or if TYPE is SQL_TABLE_STAT.
CARDINALITY	INTEGER	Cardinality of table or index; number of rows in table if TYPE is SQL_TABLE_STAT; number of unique values in the index if TYPE is not SQL_TABLE_STAT; NULL is returned if the value is not available from the data source.
PAGES	INTEGER	Number of pages used to store the index or table; number of pages for the table if TYPE is SQL_TABLE_STAT; number of pages for the index if TYPE is not SQL_TABLE_STAT; NULL is returned if the value is not available from the data source, or if not applicable to the data source.
FILTER_CONDITION	VARHAR(128)	If the index is a filtered index, this is the filter condition, such as SALARY > 30000; if the filter condition cannot be determined, this is an empty string. NULL if the index is not a filtered index, it cannot be determined whether the index is a filtered index, or TYPE is SQL_TABLE_STAT.

(3 of 3)

If the row in the result set corresponds to a table, the driver sets TYPE to SQL_TABLE_STAT and sets NON_UNIQUE, INDEX_QUALIFIER, INDEX_NAME, SEQ_IN_INDEX, COLUMN_NAME, and COLLATION to NULL. If CARDINALITY or PAGES are not available from the data source, the driver sets them to NULL.

Code Example

For a code example of a similar function, see **SQLColumns**.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning the columns of foreign keys	**SQLForeignKeys**
Returning the columns of a primary key	**SQLPrimaryKeys**

♦

Level 2

SQLTablePrivileges

SQLTablePrivileges returns a list of tables and the privileges associated with each table. The driver returns the information as a result set on the specified *hstmt*.

Syntax

```
RETCODE SQLTablePrivileges(hstmt, szTableQualifier,
cbTableQualifier,
szTableOwner, cbTableOwner, szTableName, cbTableName)
```

The **SQLTablePrivileges** function accepts the following arguments.

Typedef	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szTableQualifier*	Input	Table qualifier. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbTableQualifier*	Input	Length of *szTableQualifier.*
UCHAR FAR *	*szTableOwner*	Input	String search pattern for owner names. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have owners.
SWORD	*cbTableOwner*	Input	Length of *szTableOwner.*
UCHAR FAR *	*szTableName*	Input	String search pattern for table names.
SWORD	*cbTableName*	Input	Length of *szTableName.*

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver was unable to allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

(1 of 2)

SQLSTATE	Error	Description
S1090	Invalid string or buffer length	(DM) The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS. The value of one of the name-length arguments exceeded the maximum-length value for the corresponding qualifier or name.
S1C00	Driver not capable	A table qualifier was specified, but the driver or data source does not support qualifiers. A table owner was specified, but the driver or data source does not support owners. A string search pattern was specified for the table owner, table name, or column name and the data source does not support search patterns for one or more of those arguments. The driver or data source does not support the combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options.
S1T00	Time-out expired	The time-out period expired before the data source returned the result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

The *szTableOwner* and *szTableName* arguments accept search patterns. For more information about valid search patterns, see "Search Pattern Arguments" on page 12-8.

SQLTablePrivileges returns the results as a standard result set, ordered by TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, and PRIVILEGE. The following table lists the columns in the result set.

Important: ***SQLTablePrivileges*** *might not return privileges for all tables. Applications can use any valid table, regardless of whether* ***SQLTablePrivileges*** *returns it.*

The lengths of VARCHAR columns shown in the table are maximums; the actual lengths depend on the data source. To determine the actual lengths of the TABLE_QUALIFIER, TABLE_OWNER, and TABLE_NAME columns, an application can call **SQLGetInfo** with the SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, and SQL_MAX_TABLE_NAME_LEN options.

Column Name	Informix SQL Data Type	Comments
TABLE_QUALIFIER	VARCHAR(128)	Table-qualifier identifier; NULL if not applicable to the data source. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have qualifiers.
TABLE_OWNER	VARCHAR(128)	Table-owner identifier; NULL if not applicable to the data source. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have owners.
TABLE_NAME	VARCHAR(128) not NULL	Table identifier.
GRANTOR	VARCHAR(128)	Identifier of the user who granted the privilege; NULL if not applicable to the data source.
GRANTEE	VARCHAR(128) not NULL	Identifier of the user to whom the privilege was granted.

(1 of 2)

Column Name	Informix SQL Data Type	Comments
PRIVILEGE	VARCHAR(128) not NULL	Identifies the table privilege. Can be one of the following or a data-source-specific privilege: ■ SELECT: The grantee is permitted to retrieve data for one or more columns of the table. ■ INSERT: The grantee is permitted to insert new rows containing data for one or more columns into to the table. ■ UPDATE: The grantee is permitted to update the data in one or more columns of the table. ■ DELETE: The grantee is permitted to delete rows of data from the table. ■ REFERENCES: The grantee is permitted to refer to one or more columns of the table within a constraint (for example, a unique, referential, or table-check constraint). The scope of action permitted the grantee by a given table privilege is data source–dependent. For example, the UPDATE privilege might permit the grantee to update all columns in a table on one data source and only those columns for which the grantor has the UPDATE privilege on another data source.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users; “YES”, “NO”, or NULL if unknown or not applicable to the data source.

(2 of 2)

Code Example

For a code example of a similar function, see **SQLColumns**.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning privileges for a column or columns	**SQLColumnPrivileges**
Returning the columns in a table or tables	**SQLColumns**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning table statistics and indexes	**SQLStatistics**
Returning a list of tables in a data source	**SQLTables**

♦

Level 1

SQLTables

SQLTables returns the list of table names that are stored in a specific data source. The driver returns this information as a result set.

Syntax

```
RETCODE SQLTables(hstmt, szTableQualifier, cbTableQualifier,
szTableOwner, cbTableOwner, szTableName, cbTableName,
szTableType, cbTableType)
```

The **SQLTables** function accepts the following arguments.

Typedef	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle for retrieved results.
UCHAR FAR *	*szTableQualifier*	Input	Qualifier name. If a driver supports qualifiers for some tables but not for others, such as when a driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have qualifiers.
SWORD	*cbTableQualifier*	Input	Length of *szTableQualifier.*
UCHAR FAR *	*szTableOwner*	Input	String search pattern for owner names.
SWORD	*cbTableOwner*	Input	Length of *szTableOwner.*
UCHAR FAR *	*szTableName*	Input	String search pattern for table names. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have owners.
SWORD	*cbTableName*	Input	Length of *szTableName.*
UCHAR FAR *	*szTableType*	Input	List of table types to match.
SWORD	*cbTableType*	Input	Length of *szTableType.*

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication-link failure	The communication link between the driver and the data source failed before the function completed.
24000	Invalid cursor state	A cursor was already opened on the statement handle.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1008	Operation canceled	The function was called, but before it completed execution, **SQLCancel** was called on the *hstmt* from a different thread in a multithreaded application.

(1 of 2)

SQLSTATE	Error	Description
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for the *hstmt* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1090	Invalid string or buffer length	(DM) The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS. The value of one of the name-length arguments exceeded the maximum-length value for the corre-sponding qualifier or name.
S1C00	Driver not capable	A table qualifier was specified, but the driver or data source does not support qualifiers. A table owner was specified, but the driver or data source does not support owners. A string search pattern was specified for the table owner or table name, but the data source does not support search patterns for one or more of those arguments. The driver of data source does not support the combination of the current settings of the SQL_CONCURRENCY and SQL_CURSOR_TYPE statement options.
S1T00	Time-out expired	The time-out period expired before the data source returned the requested result set. The time-out period is set through **SQLSetStmtOption**, SQL_QUERY_TIMEOUT.

(2 of 2)

Usage

SQLTables lists all the tables in the requested range. A user might or might not have SELECT privileges to any of these tables. To check accessibility, an application can call **SQLGetInfo** and check the SQL_ACCESSIBLE_TABLES info value.

Otherwise, the application must handle situations where the user selects a table for which SELECT privileges are not granted.

The *szTableOwner* and *szTableName* arguments accept search patterns. For more information about valid search patterns, see "Search Pattern Arguments" on page 12-8.

To support enumeration of qualifiers, owners, and table types, **SQLTables** defines the following special semantics for the *szTableQualifier, szTableOwner, szTableName,* and *szTableType* arguments:

- If *szTableQualifier* is a single percent character (%) and *szTableOwner* and *szTableName* are empty strings, the result set contains a list of valid qualifiers for the data source. (All the columns except the TABLE_QUALIFIER column contain nulls.)
- If *szTableOwner* is a single percent character (%) and *szTableQualifier* and *szTableName* are empty strings, the result set contains a list of valid owners for the data source. (All the columns except the TABLE_OWNER column contain nulls.)
- If *szTableType* is a single percent character (%), and *szTableQualifier, szTableOwner,* and *szTableName* are empty strings, then the result set contains a list of valid table types for the data source. (All the columns except the TABLE_TYPE column contain nulls.)

If *szTableType* is not a percent character or an empty string, it must contain a list of comma-separated values for the table types of interest. Each value in the list must be in single quotes (' ') or unquoted. The following two lists provide an example of these two different formats:

```
'TABLE','VIEW'
TABLE, VIEW
```

If the data source does not support a specified table type, **SQLTables** does not return any results for that type.

SQLTables returns the results as a standard result set, ordered by TABLE_TYPE, TABLE_QUALIFIER, TABLE_OWNER, and TABLE_NAME. The following table lists the columns in the result set.

Important: ***SQLTables*** *might not return all qualifiers, owners, or tables. Applications can use any valid qualifier, owner, or table, regardless of whether* ***SQLTables*** *returns it.*

The lengths of VARCHAR columns, as the following table shows, are maximums; the actual lengths depend on the data source. To determine the actual lengths of the TABLE_QUALIFIER, TABLE_OWNER, and TABLE_NAME columns, an application can call **SQLGetInfo** with the SQL_MAX_QUALIFIER_NAME_LEN, SQL_MAX_OWNER_NAME_LEN, and SQL_MAX_TABLE_NAME_LEN options.

Column Name	Informix SQL Data Type	Comments
TABLE_QUALIFIER	VARCHAR(128)	Table-qualifier identifier; NULL if not applicable to the data source. If a driver supports qualifiers for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have qualifiers.
TABLE_OWNER	VARCHAR(128)	Table-owner identifier; NULL if not applicable to the data source. If a driver supports owners for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have owners.
TABLE_NAME	VARCHAR(128)	Table identifier.
TABLE_TYPE	VARCHAR(128)	Table type; one of the following: “TABLE,” “VIEW,” “SYSTEM TABLE,” “GLOBAL TEMPORARY,” “LOCAL TEMPORARY,” “ALIAS,” “SYNONYM,” or a data-source-specific table type.
REMARKS	VARCHAR(254)	A description of the table.

Code Example

For a code example of a similar function, see **SQLColumns**.

Related Functions

For Information About	See
Assigning storage for a column in a result set	**SQLBindCol**
Canceling statement processing	**SQLCancel**
Returning privileges for a column or columns	**SQLColumnPrivileges**
Returning the columns in a table or tables	**SQLColumns**
Fetching a block of data or scrolling through a result set	**SQLExtendedFetch**
Fetching a row of data	**SQLFetch**
Returning table statistics and indexes	**SQLStatistics**
Returning privileges for a table or tables	**SQLTablePrivileges**

♦

Core

SQLTransact

SQLTransact requests a commit or rollback operation for all active operations on all *hstmts* associated with a connection. **SQLTransact** can also request that a commit or rollback operation be performed for all connections associated with the *henv*.

Syntax

```
RETCODE SQLTransact(henv, hdbc, fType)
```

The **SQLTransact** function accepts the following arguments.

Typedef	Argument	Use	Description
HENV	*henv*	Input	Environment handle
HDBC	*hdbc*	Input	Connection handle
UWORD	*fType*	Input	One of the following two values: SQL_COMMIT SQL_ROLLBACK

Return Codes

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Diagnostics

When the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, you can call **SQLError** to get the SQLSTATE value. The following table describes the SQLSTATE values for the function. The return code for each SQLSTATE value is SQL_ERROR unless an SQLSTATE description indicates otherwise.

If you are not using a driver manager, the driver returns the SQLSTATE value. If you are using a driver manager, the notation (DM) at the beginning of an SQLSTATE description indicates that the driver manager returns the SQLSTATE value; otherwise, the driver returns the SQLSTATE value.

SQLSTATE	Error	Description
01000	General warning	INFORMIX-CLI informational message (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	(DM) The *hdbc* was not in a connected state.
08007	Connection failure during transaction	The connection associated with the *hdbc* failed during the execution of the function; it cannot be determined whether the requested COMMIT or ROLLBACK occurred before the failure.
S1000	General error	An error occurred for which no specific SQLSTATE existed and for which no implementation-specific SQLSTATE was defined. The error message that **SQLError** returns in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory-allocation failure	The driver did not allocate memory required to support execution or completion of the function.
S1010	Function-sequence error	(DM) **SQLExecute** or **SQLExecDirect** was called for an *hstmt* associated with the *hdbc* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
S1012	Invalid transaction operation code	(DM) The value specified for the argument *fType* was neither SQL_COMMIT nor SQL_ROLLBACK.
S1C00	Driver not capable	The driver or data source does not support the ROLLBACK operation.

Usage

If *hdbc* is SQL_NULL_HDBC and *henv* is a valid environment handle, the driver manager attempts to commit or roll back transactions on all connection handles that are in a connected state. The driver manager calls **SQLTransact** in the driver associated with each *hdbc*. The driver manager returns SQL_SUCCESS only if it receives SQL_SUCCESS for each *hdbc*. If the driver manager receives SQL_ERROR on one or more connection handles, it returns SQL_ERROR to the application. To determine which connection(s) failed during the COMMIT or ROLLBACK operation, the application can call **SQLError** for each *hdbc*.

Important: *The driver manager does not simulate a global transaction across all hdbcs and therefore does not use two-phase commit protocols.*

If *hdbc* is a valid connection handle, *henv* is ignored, and the driver manager calls **SQLTransact** in the driver for the *hdbc*.

If *hdbc* is SQL_NULL_HDBC and *henv* is SQL_NULL_HENV, **SQLTransact** returns SQL_INVALID_HANDLE.

If *fType* is SQL_COMMIT, **SQLTransact** issues a COMMIT request for all active operations on any *hstmt* associated with an affected *hdbc*.

If *fType* is SQL_ROLLBACK, **SQLTransact** issues a ROLLBACK request for all active operations on any *hstmt* associated with an affected *hdbc*. If no transactions are active, **SQLTransact** returns SQL_SUCCESS with no effect on any data sources.

If the driver is in manual-commit mode (by calling **SQLSetConnectOption** with the SQL_AUTOCOMMIT option set to zero), a new transaction is started implicitly when an SQL statement that can be contained within a transaction is executed against the current data source.

To determine how transaction operations affect cursors, an application calls **SQLGetInfo** with the SQL_CURSOR_ROLLBACK_BEHAVIOR and SQL_CURSOR_COMMIT_BEHAVIOR options.

If the value of SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR equals SQL_CB_DELETE, **SQLTransact** closes and deletes all open cursors on all statement handles that are associated with the *hdbc* and discards all pending results. **SQLTransact** leaves any *hstmt* present in an allocated (unprepared) state; the application can reuse them for subsequent SQL requests or can call **SQLFreeStmt** to deallocate them.

If the value of SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR equals SQL_CB_CLOSE, **SQLTransact** closes all open cursors on all *hstmts* associated with the *hdbc*. **SQLTransact** leaves any *hstmt* present in a prepared state; the application can call **SQLExecute** for an *hstmt* associated with the *hdbc* without first calling **SQLPrepare**.

If the value of SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR equals SQL_CB_PRESERVE, **SQLTransact** does not affect open cursors associated with the *hdbc*. Cursors remain at the row to which they pointed before the call to **SQLTransact**.

For drivers and data sources that support transactions, calling **SQLTransact** with either SQL_COMMIT or SQL_ROLLBACK when no transaction is active returns SQL_SUCCESS (indicating that there is no work to be committed or rolled back) and has no effect on the data source.

Drivers or data sources that do not support transactions (**SQLGetInfo** *fOption* SQL_TXN_CAPABLE is 0) are effectively always in autocommit mode. Therefore, calling **SQLTransact** with SQL_COMMIT returns SQL_SUCCESS. However, calling **SQLTransact** with SQL_ROLLBACK results in SQLSTATE S1C00 (Driver not capable), indicating that a rollback can never be performed.

Related Functions

For Information About	See
Returning information about a driver or data source	**SQLGetInfo**
Freeing a statement handle	**SQLFreeStmt**

♦

Chapter

13 Setup Shared Library Function Reference

This chapter describes the syntax of the driver-setup shared library API, which consists of the function **ConfigDSN**, which can be in the driver shared library or in a separate setup shared library.

This chapter also describes the syntax of the translator-setup shared library API, which consists of the function **ConfigTranslator,** which can be in the translator-setup shared library or in a separate-setup shared library.

For information on argument naming conventions, see "Arguments" on page 12-4.

ConfigDSN

ConfigDSN adds, modifies, or deletes data sources. It might prompt the user for connection information. It can be in the driver shared library or a separate-setup shared library.

Syntax

```
BOOL ConfigDSN(hwndParent, fRequest, lpszDriver,
lpszAttributes)
```

The **ConfigDSN** function accepts the following arguments.

Type	Argument	Use	Description
HWND	*hwndParent*	Input	Parent window handle. The function will not display any dialog boxes if the handle is null.
UINT	*fRequest*	Input	Type of request. *fRequest* must contain one of the following values: ■ ODBC_ADD_DSN Add a new data source. ■ ODBC_CONFIG_DSN Configure (modify) an existing data source. ■ ODBC_REMOVE_DSN Remove an existing data source.
LPCSTR	*lpszDriver*	Input	Driver description (usually the name of the associated DBMS) presented to users instead of the physical driver name.
LPCSTR	*lpszAttributes*	Input	List of attributes in the form of keyword-value pairs. For information about the list structure, see "Comments."

Returns

The function returns TRUE if it is successful. It returns FALSE if it fails.

Usage

ConfigDSN receives connection information from the installer shared library as a list of attributes in the form of keyword-value pairs. Each pair is terminated with a null byte, and the entire list is null terminated (that is, two null bytes mark the end of the list). The keywords that **ConfigDSN** uses are the same as those that **SQLBrowseConnect** and **SQLDriverConnect** use, except **ConfigDSN** does not accept the DRIVER keyword. As in **SQLBrowseConnect** and **SQLDriverConnect**, the keywords and their values should not contain the [ ]{ }( ),;?*=!@\ characters, and the value of the DSN keyword cannot consist of blanks only. Because of the registry grammar, keywords and data-source names cannot contain the backslash (\) character.

For example, to configure a data source that requires a user ID, password, and database name, a setup application might pass the following keyword-value pairs:

```
DSN=Personnel Data\0UID=Smith\0PWD=Sesame\0DATABASE=Personnel\0\0
```

For more information about these keywords, see "SQLDriverConnect" on page 12-94.

To display a dialog box, *hwndParent* must not be null.

Adding a Data Source

If a data-source name is passed to **ConfigDSN** in *lpszAttributes*, **ConfigDSN** checks that the name is valid. If the data-source name matches an existing data-source name and *hwndParent* is null, **ConfigDSN** overwrites the existing name. If it matches an existing name and *hwndParent* is not null, **ConfigDSN** prompts the user to overwrite the existing name.

If *lpszAttributes* contains enough information to connect to a data source, **ConfigDSN** can add the data source or display a dialog box with which the user can change the connection information. If *lpszAttributes* does not contain enough information to connect to a data source, **ConfigDSN** must determine the necessary information; if *hwndParent* is not null, it displays a dialog box to retrieve the information from the user.

If **ConfigDSN** displays a dialog box, it must display any connection information passed to it in *lpszAttributes*. In particular, if a data-source name was passed to it, **ConfigDSN** displays that name but does not allow the user to change it. **ConfigDSN** can supply default values for connection information not passed to it in *lpszAttributes*.

If **ConfigDSN** cannot get complete connection information for a data source, it returns FALSE.

If **ConfigDSN** can get complete connection information for a data source, it calls **SQLWriteDSNToIni** in the installer shared library to add the new data-source specification. **SQLWriteDSNToIni** adds the data-source name to the ODBC Data Sources section, creates the data-source-specification section, and adds the **Driver** keyword with the driver description as its value. **ConfigDSN** calls **SQLWritePrivateProfileString** in the installer shared library to add more keywords and values that the driver needs.

Modifying a Data Source

To modify a data source, a data-source name must be passed to **ConfigDSN** in *lpszAttributes*.

If *hwndParent* is null, **ConfigDSN** uses the information in *lpszAttributes*. If *hwndParent* is not null, **ConfigDSN** displays a dialog box that uses the information in *lpszAttributes*. The user can modify the information before **ConfigDSN** stores it.

If the data-source name was changed, **ConfigDSN** first calls **SQLRemoveDSNFromIni** in the installer shared library to remove the existing data-source specification. It then follows the steps in the previous section to add the new data-source specification. If the data-source name was not changed, **ConfigDSN** calls **SQLWritePrivateProfileString** in the installer shared library to make any other changes. **ConfigDSN** cannot delete or change the value of the **Driver** keyword.

Deleting a Data Source

To delete a data source, a data-source name must be passed to **ConfigDSN** in *lpszAttributes*. **ConfigDSN** then calls **SQLRemoveDSNFromIni** in the installer shared library to remove the data source.

ConfigTranslator

ConfigTranslator returns a default translation option for a translator. It can be in the translator shared library or a separate setup shared library.

Syntax

```
BOOL ConfigTranslator(hwndParent, pvOption)
```

The **ConfigTranslator** function accepts the following arguments.

Type	Argument	Use	Description
HWND	*hwndParent*	Input	Parent window handle. The function will not display any dialog boxes if the handle is null.
DWORD FAR *	*pvOption*	Output	A 32-bit translation option.

Returns

The function returns TRUE if it is successful. It returns FALSE if it fails.

Comments

If the translator supports only a single translation option, **ConfigTranslator** returns TRUE and sets *pvOption* to the 32-bit option. Otherwise, it determines the default translation option to use. **ConfigTranslator** can display a dialog box with which a user selects a default translation option.

Related Functions

For information about	See
Getting a translation option	**SQLGetConnectOption**
Setting a translation option	**SQLSetConnectOption**

INFORMIX-CLI Error Codes

SQLError returns SQLSTATE values as defined by the X/Open and SQL Access Group SQL CAE specification (1992). SQLSTATE values are strings that contain five characters. The following table lists SQLSTATE values that the Informix driver can return for **SQLError**.

SQLSTATE Values

The character-string value returned for an SQLSTATE consists of a two-character class value followed by a three-character subclass value. A class value of `01` indicates a warning and is accompanied by a return code of SQL_SUCCESS_WITH_INFO. Class values other than `01`, except for the class `IM`, indicate an error and are accompanied by a return code of SQL_ERROR. The class `IM` is specific to warnings and errors that derive from the implementation of INFORMIX-CLI. The subclass value `000` in any class is for implementation-defined conditions within the given class. ANSI SQL-92 defines the assignment of class and subclass values.

A return value of SQL_SUCCESS normally indicates a function has executed successfully, although the SQLSTATE 00000 also indicates success.

SQL STATE	Error	Can be returned from
01000	General warning	All functions except: **SQLAllocEnv** **SQLError**
01002	Disconnect error	**SQLDisconnect**
01004	Data truncated	**SQLBrowseConnect** **SQLColAttributes** **SQLDataSources** **SQLDescribeCol** **SQLDriverConnect** **SQLDrivers** **SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLGetCursorName** **SQLGetData** **SQLGetInfo** **SQLNativeSql** **SQLPutData**
01006	Privilege not revoked	**SQLExecDirect** **SQLExecute**
01S00	Invalid connection string attribute	**SQLBrowseConnect** **SQLDriverConnect**
01S01	Error in row	**SQLExtendedFetch**
01S02	Option value changed	**SQLSetConnectOption** **SQLSetStmtOption**
01S03	No rows updated or deleted	**SQLExecDirect** **SQLExecute**
01S04	More than one row updated or deleted	**SQLExecDirect** **SQLExecute**
07001	Wrong number of parameters	**SQLExecDirect** **SQLExecute**

(1 of 13)

SQL STATE	Error	Can be returned from
07006	Restricted data-type-attribute violation	**SQLBindParameter** **SQLExtendedFetch** **SQLFetch** **SQLGetData**
08001	Unable to connect to data source	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect**
08002	Connection in use	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect** **SQLSetConnectOption**
08003	Connection not open	**SQLAllocStmt** **SQLDisconnect** **SQLGetConnectOption** **SQLGetInfo** **SQLNativeSql** **SQLSetConnectOption** **SQLTransact**
08004	Data source rejected establishment of connection	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect**
08007	Connection failure during transaction	**SQLTransact**

(2 of 13)

SQL STATE	Error	Can be returned from
08S01	Communication-link failure	**SQLBrowseConnect** **SQLColumnPrivileges** **SQLColumns** **SQLConnect** **SQLDriverConnect** **SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLForeignKeys** **SQLFreeConnect** **SQLGetData** **SQLGetTypeInfo** **SQLParamData** **SQLPrepare** **SQLPrimaryKeys** **SQLProcedureColumns** **SQLProcedures** **SQLPutData** **SQLSetConnectOption** **SQLSetStmtOption** **SQLSpecialColumns** **SQLStatistics** **SQLTablePrivileges** **SQLTables**
21S01	Insert value list does not match column list	**SQLExecDirect** **SQLPrepare**
21S02	Degree of derived table does not match column list	**SQLExecDirect** **SQLPrepare**
22001	String data right truncation	**SQLPutData**
22002	Indicator variable required but not supplied	**SQLExtendedFetch** **SQLFetch** **SQLGetData**

(3 of 13)

SQL STATE	Error	Can be returned from
22003	Numeric value out of range	**SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLGetData** **SQLGetInfo** **SQLPutData**
22005	Error in assignment	**SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLGetData** **SQLPrepare** **SQLPutData**
22008	DATETIME field overflow	**SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLGetData** **SQLPutData**
22012	Division by zero	**SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLGetData**
22026	String data, length mismatch	**SQLParamData**
23000	Integrity-constraint violation	**SQLExecDirect** **SQLExecute**

(4 of 13)

SQL STATE	Error	Can be returned from
24000	Invalid cursor state	**SQLColAttributes** **SQLColumnPrivileges** **SQLColumns** **SQLDescribeCol** **SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLForeignKeys** **SQLGetData** **SQLGetStmtOption** **SQLGetTypeInfo** **SQLPrepare** **SQLPrimaryKeys** **SQLProcedureColumns** **SQLProcedures** **SQLSetCursorName** **SQLSetStmtOption** **SQLSpecialColumns** **SQLStatistics** **SQLTablePrivileges** **SQLTables**
25000	Invalid transaction state	**SQLDisconnect**
28000	Invalid authorization specification	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect**
34000	Invalid cursor name	**SQLExecDirect** **SQLPrepare** **SQLSetCursorName**
37000	Syntax error or access violation	**SQLExecDirect** **SQLNativeSql** **SQLPrepare**
3C000	Duplicate cursor name	**SQLSetCursorName**

(5 of 13)

SQL STATE	Error	Can be returned from
40001	Serialization failure	**SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch**
42000	Syntax error or access violation	**SQLExecDirect** **SQLExecute** **SQLPrepare**
70100	Operation aborted	**SQLCancel**
IM001	Driver does not support this function	All functions except: **SQLAllocEnv** **SQLDataSources** **SQLDrivers** **SQLError** **SQLFreeConnect** **SQLFreeEnv** **SQLGetFunctions**
IM002	Data-source name not found and no default driver specified	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect**
IM003	Specified driver could not be loaded	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect**
IM004	Driver **SQLAllocEnv** failed	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect**
IM005	Driver **SQLAllocConnect** failed	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect**
IM006	Driver **SQLSetConnectOption** failed	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect**
IM007	No data source or driver specified; dialog prohibited	**SQLDriverConnect**
IM008	Dialog failed	**SQLDriverConnect**

(6 of 13)

SQL STATE	Error	Can be returned from
IM009	Unable to load translation shared library	**SQLBrowseConnect** **SQLConnect** **SQLDriverConnect** **SQLSetConnectOption**
IM010	Data-source name too long	**SQLBrowseConnect** **SQLDriverConnect**
IM011	Driver name too long	**SQLBrowseConnect** **SQLDriverConnect**
IM012	DRIVER keyword syntax error	**SQLBrowseConnect** **SQLDriverConnect**
IM013	Trace file error	All functions.
S0001	Base table or view already exists	**SQLExecDirect** **SQLPrepare**
S0002	Base table not found	**SQLExecDirect** **SQLPrepare**
S0011	Index already exists	**SQLExecDirect** **SQLPrepare**
S0012	Index not found	**SQLExecDirect** **SQLPrepare**
S0021	Column already exists	**SQLExecDirect** **SQLPrepare**
S0022	Column not found	**SQLExecDirect** **SQLPrepare**
S1000	General error	All functions except: **SQLAllocEnv** **SQLError**
S1001	Memory-allocation failure	All functions except: **SQLError** **SQLFreeConnect** **SQLFreeEnv**

(7 of 13)

SQL STATE	Error	Can be returned from
S1002	Invalid column number	**SQLBindCol** **SQLColAttributes** **SQLDescribeCol** **SQLExtendedFetch** **SQLFetch** **SQLGetData**
S1003	Program type out of range	**SQLBindCol** **SQLBindParameter** **SQLGetData**
S1004	SQL data type out of range	**SQLBindParameter** **SQLGetTypeInfo**
S1009	Invalid argument value	**SQLAllocConnect** **SQLAllocStmt** **SQLBindCol** **SQLBindParameter** **SQLExecDirect** **SQLForeignKeys** **SQLGetData** **SQLGetInfo** **SQLNativeSql** **SQLPrepare** **SQLPutData** **SQLSetConnectOption** **SQLSetCursorName** **SQLSetStmtOption**

(8 of 13)

SQL STATE	Error	Can be returned from
S1010	Function-sequence error	**SQLBindCol** **SQLBindParameter** **SQLColAttributes** **SQLColumnPrivileges** **SQLColumns** **SQLDescribeCol** **SQLDisconnect** **SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLForeignKeys** **SQLFreeConnect** **SQLFreeEnv** **SQLFreeStmt** **SQLGetConnectOption** **SQLGetCursorName** **SQLGetData** **SQLGetFunctions** **SQLGetStmtOption** **SQLGetTypeInfo** **SQLMoreResults** **SQLNumParams** **SQLNumResultCols** **SQLParamData** **SQLParamOptions** **SQLPrepare** **SQLPrimaryKeys** **SQLProcedureColumns** **SQLProcedures** **SQLPutData** **SQLRowCount** **SQLSetConnectOption** **SQLSetCursorName** **SQLSetScrollOptions** **SQLSetStmtOption** **SQLSpecialColumns** **SQLStatistics** **SQLTablePrivileges** **SQLTables** **SQLTransact**

(9 of 13)

SQL STATE	Error	Can be returned from
S1011	Operation invalid at this time	**SQLGetStmtOption** **SQLSetConnectOption** **SQLSetStmtOption**
S1012	Invalid transaction operation code specified	**SQLTransact**
S1015	No cursor name available	**SQLGetCursorName**
S1090	Invalid string or buffer length	**SQLBindCol** **SQLBindParameter** **SQLBrowseConnect** **SQLColAttributes** **SQLColumnPrivileges** **SQLColumns** **SQLConnect** **SQLDataSources** **SQLDescribeCol** **SQLDriverConnect** **SQLDrivers** **SQLExecDirect** **SQLExecute** **SQLForeignKeys** **SQLGetCursorName** **SQLGetData** **SQLGetInfo** **SQLNativeSql** **SQLPrepare** **SQLPrimaryKeys** **SQLProcedureColumns** **SQLProcedures** **SQLPutData** **SQLSetCursorName** **SQLSpecialColumns** **SQLStatistics** **SQLTablePrivileges** **SQLTables**
S1091	Descriptor type out of range	**SQLColAttributes**

(10 of 13)

SQL STATE	Error	Can be returned from
S1092	Option type out of range	**SQLFreeStmt** **SQLGetConnectOption** **SQLGetStmtOption** **SQLSetConnectOption** **SQLSetStmtOption**
S1093	Invalid parameter number	**SQLBindParameter**
S1094	Invalid scale value	**SQLBindParameter**
S1095	Function type out of range	**SQLGetFunctions**
S1096	Information type out of range	**SQLGetInfo**
S1097	Column type out of range	**SQLSpecialColumns**
S1098	Scope type out of range	**SQLSpecialColumns**
S1099	Nullable type out of range	**SQLSpecialColumns**
S1100	Uniqueness option type out of range	**SQLStatistics**
S1101	Accuracy option type out of range	**SQLStatistics**
S1103	Direction option out of range	**SQLDataSources** **SQLDrivers**
S1104	Invalid precision value	**SQLBindParameter**
S1105	Invalid parameter type	**SQLBindParameter**
S1106	Fetch type out of range	**SQLExtendedFetch**
S1107	Row value out of range	**SQLExtendedFetch** **SQLParamOptions** **SQLSetScrollOptions**
S1108	Concurrency option out of range	**SQLSetScrollOptions**
S1109	Invalid cursor position	**SQLExecute** **SQLExecDirect** **SQLGetData** **SQLGetStmtOption**
S1110	Invalid driver completion	**SQLDriverConnect**

(11 of 13)

SQL STATE	Error	Can be returned from
S1111	Invalid bookmark value	**SQLExtendedFetch**
S1C00	Driver not capable	**SQLBindCol** **SQLBindParameter** **SQLColAttributes** **SQLColumnPrivileges** **SQLColumns** **SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLForeignKeys** **SQLGetConnectOption** **SQLGetData** **SQLGetInfo** **SQLGetStmtOption** **SQLGetTypeInfo** **SQLPrepare** **SQLPrimaryKeys** **SQLProcedureColumns** **SQLProcedures** **SQLSetConnectOption** **SQLSetScrollOptions** **SQLSetStmtOption** **SQLSpecialColumns** **SQLStatistics** **SQLTablePrivileges** **SQLTables** **SQLTransact**

(12 of 13)

SQL STATE	Error	Can be returned from
S1T00	Time-out expired	**SQLBrowseConnect** **SQLColAttributes** **SQLColumnPrivileges** **SQLColumns** **SQLConnect** **SQLDescribeCol** **SQLDriverConnect** **SQLExecDirect** **SQLExecute** **SQLExtendedFetch** **SQLFetch** **SQLForeignKeys** **SQLGetData** **SQLGetInfo** **SQLGetTypeInfo** **SQLMoreResults** **SQLNumParams** **SQLNumResultCols** **SQLParamData** **SQLPrepare** **SQLPrimaryKeys** **SQLProcedureColumns** **SQLProcedures** **SQLPutData** **SQLSpecialColumns** **SQLStatistics** **SQLTablePrivileges** **SQLTables**

(13 of 13)

Data Types

This appendix discusses the following topics:

- Types of data types
- SQL data types
- C data types
- Transferring data
- Converting data

Types of Data Types

The following table describes the types of data types that INFORMIX-CLI supports.

Type of Data Type	Description	Example
Informix SQL data type	Data types that your Informix database server uses	CHAR(*n*)
INFORMIX-CLI SQL data type	Data types that correspond to the Informix SQL data types	SQL_CHAR
Standard C data type	Data types that your C compiler defines	unsigned char
Typedef for a standard C data type	Typedefs that correspond to the standard C data types	UCHAR
INFORMIX-CLI C data type	Data types that correspond to the standard C data types	SQL_C_CHAR

SQL Data Types

For detailed information about the Informix SQL data types, see the *Informix Guide to SQL: Reference*.

Standard SQL Data Types

The following table lists the standard Informix SQL data types and their corresponding INFORMIX-CLI data types.

Informix SQL Data Type	INFORMIX-CLI SQL Data Type (fSqlType)	Description
BYTE	SQL_LONGVARBINARY	Binary data of variable length
CHAR(*n*), CHARACTER(*n*)	SQL_CHAR	Character string of fixed length *n* ($1 \le n \le 32{,}767$)
CHARACTER VARYING(*m*, *r*)	SQL_VARCHAR	Character string of variable length with maximum length *m* ($1 \le m \le 255$) and minimum amount of reserved space *r* ($0 \le r < m$)
DATE	SQL_DATE	Calendar date
DATETIME	SQL_TIMESTAMP	Calendar date and time of day
DEC(*p*, *s*), DECIMAL(*p*, *s*)	SQL_DECIMAL	Signed numeric value with precision *p* and scale *s* ($1 \le p \le 32$; $0 \le s \le p$)
DOUBLE PRECISION	SQL_DOUBLE	Signed numeric value with the same characteristics as the **double** data type in C
FLOAT	SQL_DOUBLE	Signed numeric value with the same characteristics as the **double** data type in C
INT, INTEGER	SQL_INTEGER	Signed numeric value with precision 10, scale 0, and range *n* ($-2{,}147{,}483{,}647 \le n \le 2{,}147{,}483{,}647$)
MONEY(*p*, *s*)	SQL_DECIMAL	Signed numeric value with precision *p* and scale *s* ($1 \le p \le 32$; $0 \le s \le p$)
NUMERIC	SQL_DECIMAL	Signed numeric value with precision *p* and scale *s* ($1 \le p \le 32$; $0 \le s \le p$)

(1 of 2)

Informix SQL Data Type	**INFORMIX-CLI SQL Data Type (fSqlType)**	**Description**
REAL	SQL_REAL	Signed numeric value with the same characteristics as the **float** data type in C
SERIAL	SQL_INTEGER	Sequential INTEGER
SMALLFLOAT	SQL_REAL	Signed numeric value with the same characteristics as the **float** data type in C
SMALLINT	SQL_SMALLINT	Signed numeric value with precision 5, scale 0, and range *n* ($-32,767 \le n \le 32,767$)
TEXT	SQL_LONGVARCHAR	Character string of variable length
VARCHAR(*m*, *r*)	SQL_VARCHAR	Character string of variable length with maximum length *m* ($1 \le m \le 255$) and minimum amount of reserved space *r* ($0 \le r < m$)

(2 of 2)

GLS

Additional SQL Data Types for GLS

The following table lists the additional Informix SQL data types for GLS and their corresponding INFORMIX-CLI data types. INFORMIX-CLI does not provide full GLS support.

Informix SQL Data Type	**INFORMIX-CLI SQL Data Type (fSqlType)**	**Description**
NCHAR(*n*)	SQL_CHAR	Character string of fixed length *n* ($1 \le n \le 32,767$). Collation depends on locale.
NVARCHAR(*m*, *r*)	SQL_VARCHAR	Character string of variable length with maximum length *m* ($1 \le m \le 255$) and minimum amount of reserved space *r* ($0 \le r < m$). Collation depends on locale.

♦

IUS

Additional SQL Data Types for Universal Server

The following table lists the additional Informix SQL data types for Universal Server and their corresponding INFORMIX-CLI data types.

Informix SQL Data Type	INFORMIX-CLI SQL Data Type (fSqlType)	Description
BOOLEAN	SQL_BIT	't' or 'f'
DISTINCT	DISTINCT can correspond to any INFORMIX-CLI SQL data type. For more information about DISTINCT, see the *Informix Guide to SQL: Tutorial*.	UDT that is stored the same way as its source data type but has different casts and functions
INT8	SQL_BIGINT	Signed numeric value with precision 10, scale 0, and range n $(-(2^{63} - 1) \leq n \leq 2^{63} - 1)$
LVARCHAR	SQL_LONGVARCHAR	Character string of variable length
OPAQUE (fixed)	SQL_INFX_UDT_FIXED	Fixed-length UDT with an internal structure that the database server cannot access
OPAQUE (varying)	SQL_INFX_UDT_VARYING	Variable-length UDT with an internal structure that the database server cannot access
SERIAL8	SQL_BIGINT	Sequential INT8

To use the INFORMIX-CLI SQL data types for Universal Server, include **infxcli.h**. ♦

Precision, Scale, Length, and Display Size

The functions that get and set precision, scale, length, and display size for SQL values have size limitations for their input arguments. Therefore, these values are limited to the size of an SDWORD that has a maximum value of 2,147,483,647. The following table describes these values.

Value	Description for a Numeric Data Type	Description for a Non-Numeric Data Type
Precision	Maximum number of digits.	Either the maximum length or the specified length.
Scale	Maximum number of digits to the right of the decimal point. For floating point values, the scale is undefined because the number of digits to the right of the decimal point is not fixed.	Not applicable.
Length	Maximum number of bytes that a function returns when a value is transferred to its default C data type.	Maximum number of bytes that a function returns when a value is transferred to its default C data type. The length does not include the NULL termination byte.
Display size	Maximum number of bytes needed to display data in character form.	Maximum number of bytes needed to display data in character form.

Standard SQL Data Types

The following table describes the precision, scale, length, and display size for the standard INFORMIX-CLI SQL data types.

INFORMIX-CLI SQL Data Type (fSqlType)	Description
SQL_CHAR	Precision: Same as the length. Scale: Not applicable. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: The specified length. For example, the length of CHAR(10) is 10 bytes. Display size: Same as the length.
SQL_DATE	Precision: 10. **SQLBindParameter** ignores the value of *cbColDef* for this data type. Scale: Not applicable. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: 6 bytes. Display size: 10 digits in the format yyyy-mm-dd.
SQL_DECIMAL	Precision: The specified precision. For example, the precision of DECIMAL (12, 3) is 12. Scale: The specified scale. For example, the scale of DECIMAL(12, 3) is 3. Length: The specified precision plus 2. For example, the length of DECIMAL(12, 3) is 14 bytes. The two additional bytes are used for the sign and the decimal points because functions return this data type as a character string. Display size: Same as the length.
SQL_DOUBLE	Precision: 15. **SQLBindParameter** ignores the value of *cbColDef* for this data type. Scale: Not applicable. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: 8 bytes. Display size: 22 digits. The digits are for a sign, 15 numeric characters, a decimal point, the letter E, another sign, and 2 more numeric characters.

(1 of 3)

INFORMIX-CLI SQL Data Type (fSqlType)	Description
SQL_INTEGER	Precision: 10. **SQLBindParameter** ignores the value of *cbColDef* for this data type. Scale: 0. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: 4 bytes. Display size: 11 digits. One digit is for the sign.
SQL_LONGVARBINARY	Precision: Same as the length. Scale: Not applicable. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: The maximum length. If a function cannot determine the maximum length, it returns SQL_NO_TOTAL. Display size: The maximum length times 2. If a function cannot determine the maximum length, it returns SQL_NO_TOTAL.
SQL_LONGVARCHAR	Precision: Same as the length. Scale: Not applicable. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: The maximum length. If a function cannot determine the maximum length, it returns SQL_NO_TOTAL. Display size: Same as the length.
SQL_REAL	Precision: 7. **SQLBindParameter** ignores the value of *cbColDef* for this data type. Scale: Not applicable. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: 4 bytes. Display size: 13 digits. The digits are for a sign, 7 numeric characters, a decimal point, the letter E, another sign, and 2 more numeric characters.

(2 of 3)

INFORMIX-CLI SQL Data Type (fSqlType)	Description
SQL_SMALLINT	Precision: 5. **SQLBindParameter** ignores the value of *cbColDef* for this data type. Scale: 0. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: 2 bytes. Display size: 6 digits. One digit is for the sign.
SQL_TIMESTAMP	Precision: 8. **SQLBindParameter** ignores the value of *cbColDef* for this data type. Scale: The number of digits in the FRACTION field. Length: 16 bytes. Display size: 19 or more digits: ■ If the scale of the time stamp is 0: 19 digits in the format yyyy-mm-dd hh:mm:ss. ■ If the scale of the time stamp exceeds 0: 20 digits plus digits for the FRACTION field in the format yyyy-mm-dd hh:mm:ss.f...
SQL_VARCHAR	Precision: Same as the length. Scale: Not applicable. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: The specified length. For example, the length of VARCHAR(10) is 10 bytes. Display size: Same as the length.

(3 of 3)

IUS

Additional SQL Data Types for Universal Server

The following table describes the precision, scale, length, and display size for the INFORMIX-CLI SQL data types for Universal Server.

INFORMIX-CLI SQL Data Type (fSqlType)	Description
SQL_BIGINT	Precision: 19. **SQLBindParameter** ignores the value of *cbColDef* for this data type. Scale: 0. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: 8 bytes. Display size: 20 digits. One digit is for the sign.
SQL_BIT	Precision: 1. **SQLBindParameter** ignores the value of *cbColDef* for this data type. Scale: 0. **SQLBindParameter** ignores the value of *ibScale* for this data type. Length: 1 byte. Display size: 1 digit.
SQL_INFX_UDT_FIXED	Precision: Variable value. To determine this value, call a function that returns the precision for a column. Scale: Not applicable. A function that returns the scale for a column returns -1 for this data type. Length: Variable value. To determine this value, call a function that returns the length for a column. Display Size: Variable value. To determine this value, call a function that returns the display size for a column.
SQL_INFX_UDT_VARYING	Precision: Variable value. To determine this value, call a function that returns the precision for a column. Scale: Not applicable. A function that returns the scale for a column returns -1 for this data type. Length: Variable value. To determine this value, call a function that returns the length for a column. Display Size: Variable value. To determine this value, call a function that returns the display size for a column.
SQL_LONGVARCHAR	See the description in the table for the standard SQL data types.

♦

C Data Types

An INFORMIX-CLI application uses C data types to store values that the application processes.

Character C Data Type

The following table lists the character C data type that INFORMIX-CLI provides. It also lists the corresponding INFORMIX-CLI typedef and standard C data type.

INFORMIX-CLI C Data Type (fCType)	INFORMIX-CLI Typedef	Standard C Data Type
SQL_C_CHAR	UCHAR FAR *	**unsigned char FAR ***

Important: *String arguments in INFORMIX-CLI functions are unsigned. Therefore, you need to cast a CString object as an unsigned string before you use it as an argument in an INFORMIX-CLI function.*

Numeric C Data Types

The following table lists the numeric C data types that INFORMIX-CLI provides. It also lists the corresponding INFORMIX-CLI typedefs and standard C data types.

INFORMIX-CLI C Data Type (fCType)	INFORMIX-CLI Typedef	Standard C Data Type
SQL_C_DOUBLE	SDOUBLE	**signed double**
SQL_C_FLOAT	SFLOAT	**signed float**
SQL_C_LONG	SDWORD	**signed long int**
SQL_C_SHORT	SWORD	**signed short int**
SQL_C_SLONG	SDWORD	**signed long int**

(1 of 2)

INFORMIX-CLI C Data Type (fCType)	INFORMIX-CLI Typedef	Standard C Data Type
SQL_C_SSHORT	SWORD	**signed short int**
SQL_C_SS	SCHAR	**signed char**
SQL_C_TINYINT	SCHAR	**signed char**
SQL_C_ULONG	UDWORD	**unsigned long int**
SQL_C_USHORT	UWORD	**unsigned short int**
SQL_C_UTINYINT	UCHAR	**unsigned char**

(2 of 2)

Date and Time Stamp C Data Types

The following table lists the date and time stamp C data types that INFORMIX-CLI provides. It also lists the corresponding INFORMIX-CLI typedefs and standard C data types.

INFORMIX-CLI C Data Type (fCType)	INFORMIX-CLI Typedef	Standard C Data Type
SQL_C_DATE	DATE_STRUCT	`struct tagDATE_STRUCT { SWORD year; UWORD month; UWORD day; }`
SQL_C_TIMESTAMP	TIMESTAMP_STRUCT	`struct tagTIMESTAMP_STRUCT { SWORD year; UWORD month; UWORD day; UWORD hour; UWORD minute; UWORD second; UDWORD fraction; }`

Binary C Data Type

The following table lists the binary C data type that INFORMIX-CLI provides. It also lists the corresponding INFORMIX-CLI typedef and standard C data type.

INFORMIX-CLI C Data Type (fCType)	INFORMIX-CLI Typedef	Standard C Data Type
SQL_C_BINARY	UCHAR FAR *	**unsigned char FAR ***

IUS

Additional C Data Type for Universal Server

The following table lists the additional C data type that INFORMIX-CLI provides for Universal Server. It also lists the corresponding INFORMIX-CLI typedef and standard C data type.

INFORMIX-CLI C Data Type (fCType)	INFORMIX-CLI Typedef	Standard C Data Type
SQL_C_BIT	UCHAR	**unsigned char**

To use the INFORMIX-CLI C data types for Universal Server, include **infxcli.h**. ♦

Transferring Data

Among data sources that use the same DBMS, an application can safely transfer data in the internal form used by that DBMS. For a particular piece of data, the SQL data types must be the same in the source and target data sources. The C data type is SQL_C_BINARY.

When the application calls **SQLFetch**, **SQLExtendedFetch**, or **SQLGetData** to retrieve the data from the *source* data source, the driver retrieves the data from the data source and transfers it, without conversion, to a storage location of type SQL_C_BINARY. When the application calls **SQLExecute**, **SQLExecDirect**, or **SQLPutData** to send the data to the *target* data source, the driver retrieves the data from the storage location and transfers it, without conversion, to the target data source.

The binary representation of INT8 and SERIAL8 is an array of two unsigned long integers followed by a short integer that indicates the sign field. The sign field is `1` for a positive value, `-1` for a negative value, or `0` for a null value.

Important: *Applications that transfer any data (except binary data) in this manner are not interoperable among DBMSs.*

Converting Data

The word *convert* is used in this section in a broad sense; it includes the transfer of data from one storage location to another without a conversion in data type.

The following table shows the supported conversions between the Informix SQL data types and the INFORMIX-CLI C data types. A solid circle (●) indicates a supported conversion.

	INFORMIX-CLI C Data Type (fCType)														
Informix SQL Data Type	SQL_C_BINARY	SQL_C_CHAR	SQL_C_DATE	SQL_C_DOUBLE	SQL_C_FLOAT	SQL_C_LONG	SQL_C_SHORT	SQL_C_SLONG	SQL_C_SSHORT	SQL_C_STINYINT	SQL_C_TIMESTAMP	SQL_C_TINYINT	SQL_C_ULONG	SQL_C_USHORT	SQL_C_UTINYINT
BYTE	●	●													
CHAR, CHARACTER	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
CHARACTER VARYING	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
DATE	●	●	●								●				
DATETIME	●	●	●								●				
DEC, DECIMAL	●	●		●	●	●	●	●	●	●		●	●	●	●
DOUBLE PRECISION	●	●		●	●	●	●	●	●	●		●	●	●	●
FLOAT	●	●		●	●	●	●	●	●	●		●	●	●	●
INT, INTEGER	●	●		●	●	●	●	●	●	●		●	●	●	●
MONEY	●	●		●	●	●	●	●	●	●		●	●	●	●
NUMERIC	●	●		●	●	●	●	●	●	●		●	●	●	●
REAL	●	●		●	●	●	●	●	●	●		●	●	●	●

(1 of 2)

Informix SQL Data Type	SQL_C_BINARY	SQL_C_CHAR	SQL_C_DATE	SQL_C_DOUBLE	SQL_C_FLOAT	SQL_C_LONG	SQL_C_SHORT	SQL_C_SLONG	SQL_C_SSHORT	SQL_C_STINYINT	SQL_C_TIMESTAMP	SQL_C_TINYINT	SQL_C_ULONG	SQL_C_USHORT	SQL_C_UTINYINT
SERIAL	●	●		●	●	●	●	●	●	●		●	●	●	●
SMALLFLOAT	●	●		●	●	●	●	●	●	●		●	●	●	●
SMALLINT	●	●		●	●	●	●	●	●	●		●	●	●	●
TEXT	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
VARCHAR	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

INFORMIX-CLI C Data Type (fCType)

(2 of 2)

GLS

Additional Conversions for GLS

The following table shows the supported conversions between the additional Informix SQL data types for GLS and the INFORMIX-CLI C data types. A solid circle (●) indicates a supported conversion.

INFORMIX-CLI C Data Type (fCType)

Informix SQL Data Type	SQL_C_BINARY	SQL_C_CHAR	SQL_C_DATE	SQL_C_DOUBLE	SQL_C_FLOAT	SQL_C_LONG	SQL_C_SHORT	SQL_C_SLONG	SQL_C_SSHORT	SQL_C_STINYINT	SQL_C_TIMESTAMP	SQL_C_TINYINT	SQL_C_ULONG	SQL_C_USHORT	SQL_C_UTINYINT
NCHAR	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
NVARCHAR	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

♦

IUS

Additional Conversions for Universal Server

The following table shows the supported conversions between the additional Informix SQL data types for Universal Server and the INFORMIX-CLI C data types, including the additional INFORMIX-CLI C data type for Universal Server. A solid circle (●) indicates a supported conversion.

	INFORMIX-CLI C Data Type (fCType)															
Informix SQL Data Type	SQL_C_BINARY	SQL_C_BIT	SQL_C_CHAR	SQL_C_DATE	SQL_C_DOUBLE	SQL_C_FLOAT	SQL_C_LONG	SQL_C_SHORT	SQL_C_SLONG	SQL_C_SSHORT	SQL_C_STINYINT	SQL_C_TIMESTAMP	SQL_C_TINYINT	SQL_C_ULONG	SQL_C_USHORT	SQL_C_UTINYINT
BOOLEAN	●	●	●													
DISTINCT	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
INT8	●		●				●		●					●		
LVARCHAR	●		●													
OPAQUE [a]	●		●													
SERIAL8	●		●				●		●					●		

[a] Use SQL_C_CHAR to access an OPAQUE value in the external format as a string. Use SQL_C_BINARY to access an OPAQUE value in the internal binary format.

The following table shows the supported conversions between the Informix SQL data types and the additional INFORMIX-CLI C data type for Universal Server. A solid circle (●) indicates a supported conversion.

	INFORMIX-CLI C Data Type (fCType)
Informix SQL Data Type	SQL_C_BIT
BYTE	
CHAR, CHARACTER	●
CHARACTER VARYING	●
DATE	
DATETIME	
DEC, DECIMAL	●
DOUBLE PRECISION	●
FLOAT	●
INT, INTEGERS	●
MONEY	●
NUMERIC	●
REAL	●
SERIAL	●
SMALLFLOAT	●
SMALLINT	●
TEXT	●
VARCHAR	●

♦

GLS

IUS

Additional Conversions for GLS and Universal Server

The following table shows the supported conversions between the additional Informix SQL data types for GLS and the additional INFORMIX-CLI C data type for Universal Server. A solid circle (●) indicates a supported conversion.

Informix SQL Data Type	INFORMIX-CLI C Data Type (fCType) SQL_C_BIT
NCHAR	●
NVARCHAR	●

♦

Default C Data Types

If an application specifies SQL_C_DEFAULT for the *fCType* argument in **SQLBindCol**, **SQLGetData**, or **SQLBindParameter**, the driver assumes that the C data type of the output or input buffer corresponds to the SQL data type of the column or parameter to which the buffer is bound. For each INFORMIX-CLI SQL data type, the following table shows the default C data type.

INFORMIX-CLI SQL Data Type (fSqlType)	Default INFORMIX-CLI C Data Type (fCType)
SQL_CHAR	SQL_C_CHAR
SQL_DATE	SQL_C_DATE
SQL_DECIMAL	SQL_C_CHAR
SQL_DOUBLE	SQL_C_DOUBLE
SQL_INTEGER	SQL_C_SLONG
SQL_LONGVARBINARY	SQL_C_BINARY
SQL_LONGVARCHAR	SQL_C_CHAR

(1 of 2)

INFORMIX-CLI SQL Data Type (fSqlType)	Default INFORMIX-CLI C Data Type (fCType)
SQL_REAL	SQL_C_FLOAT
SQL_SMALLINT	SQL_C_SSHORT
SQL_TIMESTAMP	SQL_C_TIMESTAMP
SQL_VARCHAR	SQL_C_CHARS

(2 of 2)

IUS

Additional Default C Data Types for Universal Server

For each additional INFORMIX-CLI SQL data type for Universal Server, the following table shows the default C data type.

INFORMIX-CLI SQL Data Type (fSqlType)	Default INFORMIX-CLI C Data Type (fCType)
SQL_BIGINT	SQL_C_CHAR
SQL_BIT	SQL_C_BITS
SQL_INFX_UDT_FIXED	None [a]
SQL_INFX_UDT_VARYING	None [a]

[a] This INFORMIX-CLI SQL data type does not have a default INFORMIX-CLI C data type. Since this INFORMIX-CLI SQL data type can contain binary data or character data, you must bind a variable for this INFORMIX-CLI SQL data type before you fetch a corresponding value. The data type of the bound variable specifies the C data type for the value.

♦

Converting Data from SQL to C

When an application calls **SQLExtendedFetch**, **SQLFetch**, or **SQLGetData**, the driver retrieves the data from the data source. If necessary, it converts the data from the data type in which the driver retrieved it to the data type specified by the *fCType* argument in **SQLBindCol** or **SQLGetData**. Finally, it stores the data in the location pointed to by the *rgbValue* argument in **SQLBindCol** or **SQLGetData**.

The tables in the following sections describe how the driver or data source converts data that is retrieved from the data source; drivers are required to support conversions to all INFORMIX-CLI C data types from the INFORMIX-CLI SQL data types that they support. For a given INFORMIX-CLI SQL data type, the first column of the table lists the legal input values of the *fCType* argument in **SQLBindCol** and **SQLGetData**. The second column lists the outcomes of a test, often using the *cbValueMax* argument specified in **SQLBindCol** or **SQLGetData**, which the driver performs to determine whether it can convert the data. For each outcome, the third and fourth columns list the values of the *rgbValue* and *pcbValue* arguments specified in **SQLBindCol** or **SQLGetData** after the driver attempts to convert the data.

The last column lists the SQLSTATE returned for each outcome by **SQLExtendedFetch**, **SQLFetch**, or **SQLGetData**.

If the *fCType* argument in **SQLBindCol** or **SQLGetData** contains a value for an INFORMIX-CLI C data type that is not shown in the table for a given INFORMIX-CLI SQL data type, **SQLExtendedFetch**, **SQLFetch**, or **SQLGetData** returns SQLSTATE 07006 (Restricted data type attribute violation). If the *fCType* argument contains a value that specifies a conversion from a driver-specific SQL data type to an INFORMIX-CLI C data type and this conversion is not supported by the driver, **SQLExtendedFetch**, **SQLFetch**, or **SQLGetData** returns SQLSTATE S1C00 (Driver not capable).

Although the tables in this appendix do not show it, the *pcbValue* argument contains SQL_NULL_DATA when the SQL data value is null. When SQL data is converted to character C data, the character count returned in *pcbValue* does not include the null-termination byte. If *rgbValue* is a null pointer, **SQLBindCol** or **SQLGetData** returns SQLSTATE S1009 (Invalid argument value).

The following terms and conventions are used in the tables:

- *Length of data* is the number of bytes of C data that are available to return in *rgbValue*, regardless of whether the data is truncated before it returns to the application. For string data, this does not include the null-termination byte.
- *Display size* is the total number of bytes that are needed to display the data in character format.
- Words in *italics* represent function arguments or elements of the INFORMIX-CLI SQL grammar.

SQL-to-C: Character

The character INFORMIX-CLI SQL data types are:

- SQL_CHAR
- SQL_LONGVARCHAR
- SQL_VARCHAR

The following table shows the INFORMIX-CLI C data types to which character SQL data can be converted.

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Length of data < *cbValueMax*.	Data	Length of data	N/A
	Length of data ≥ *cbValueMax*.	Truncated data	Length of data	01004
SQL_C_LONG SQL_C_SHORT SQL_C_SLONG SQL_C_SSHORT SQL_C_STINYINT SQL_C_TINYINT SQL_C_ULONG SQL_C_USHORT SQL_C_UTINYINT	Data converted without truncation. [b]	Data	Size of the C data type	N/A
	Data converted with truncation of fractional digits. [b]	Truncated data	Size of the C data type	01004
	Conversion of data would result in loss of whole (as opposed to fractional) digits. [b]	Untouched	Untouched	22003
	Data is not a *numeric-literal.* [b]	Untouched	Untouched	22005
SQL_C_DOUBLE SQL_C_FLOAT	Data is within the range of the data type to which the number is being converted. [b]	Data	Size of the C data type	N/A
	Data is outside the range of the data type to which the number is being converted. [b]	Untouched	Untouched	22003
	Data is not a *numeric-literal.* [b]	Untouched	Untouched	22005
SQL_C_BINARY	Length of data ≤ *cbValueMax*.	Data	Length of data	N/A
	Length of data > *cbValueMax*.	Truncated data	Length of data	01004

(1 of 2)

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_DATE	Data value is a valid *date-value.* [b]	Data	6 [c]	N/A
	Data value is a valid *timestamp-value*; time portion is zero. [b]	Data	6 [c]	N/A
	Data value is a valid *timestamp-value*; time portion is non-zero.[b, d]	Truncated data	6 [c]	01004
	Data value is not a valid *date-value* or *timestamp-value.* [b]	Untouched	Untouched	22008
SQL_C_TIMESTAMP	Data value is a valid *timestamp-value*; fractional seconds portion not truncated. [b]	Data	16 [c]	N/A
	Data value is a valid *timestamp-value*; fractional seconds portion truncated. [b]	Truncated data	16 [c]	N/A
	Data value is a valid *date-value.* [b]	Data [g]	16 [c]	N/A
	Data value is a valid *time-value.* [b]	Data [h]	16 [c]	N/A
	Data value is not a valid *date-value, time-value,* or *timestamp-value.* [b]	Untouched	Untouched	22008

[b] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of rgbValue is the size of the C data type.

[c] This is the size of the corresponding C data type.

[e] The date portion of *timestamp-value* is ignored.

[f] The fractional seconds portion of the time stamp is truncated.

[g] The time fields of the time stamp structure are set to zero.

[h] The date fields of the time stamp structure are set to the current date.

(2 of 2)

When character SQL data is converted to numeric, date, or time stamp C data, leading and trailing spaces are ignored.

IUS

Additional SQL-to-C Character Data Conversion for Universal Server

The following table shows the additional INFORMIX-CLI C data type for Universal Server to which character SQL data can be converted.

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_BIT	Data is 0 or 1.	Data	1 [c]	N/A
	Data is greater than 0, less than 2, and not equal to 1.	Truncated data	1 [c]	01004
	Data is less than 0 or greater than or equal to 2.	Untouched	Untouched	22003
	Data is not a *numeric-literal*.	Untouched	Untouched	22005

[c] This is the size of the corresponding C data type.

♦

SQL-to-C: Numeric

The numeric INFORMIX-CLI SQL data types are found in the following list:

- SQL_DECIMAL
- SQL_DOUBLE
- SQL_INTEGER
- SQL_REAL
- SQL_SMALLINT

The following table shows the INFORMIX-CLI C data types to which numeric SQL data can be converted.

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < *cbValueMax*.	Data	Length of data	N/A
	Number of whole (as opposed to fractional) digits < *cbValueMax*.	Truncated data	Length of data	01004
	Number of whole (as opposed to fractional) digits ≥ *cbValueMax*.	Untouched	Untouched	22003
SQL_C_LONG SQL_C_SHORT SQL_C_SLONG SQL_C_SSHORT SQL_C_STINYINT SQL_C_TINYINT SQL_C_ULONG SQL_C_USHORT SQL_C_UTINYINT	Data converted without truncation. [b]	Data	Size of the C data type	N/A
	Data converted with truncation of fractional digits. [b]	Truncated data	Size of the C data type	01004
	Conversion of data would result in loss of whole (as opposed to fractional) digits. [b]	Untouched	Untouched	22003
SQL_C_DOUBLE SQL_C_FLOAT	Data is within the range of the data type to which the number is being converted. [b]	Data	Size of the C data type	N/A
	Data is outside the range of the data type to which the number is being converted. [b]	Untouched	Untouched	22003
SQL_C_BINARY	Length of data ≤ *cbValueMax*.	Data	Length of data	N/A
	Length of data > *cbValueMax*.	Untouched	Untouched	22003

[b] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[c] This is the size of the corresponding C data type.

IUS

Additional SQL-to-C Numeric Data Conversion for Universal Server

The additional numeric INFORMIX-CLI SQL data type is SQL_BIGINT. The previous table shows the INFORMIX-CLI C data types to which SQL_BIGINT data can be converted. The following table shows the additional INFORMIX-CLI C data type for Universal Server to which character SQL data can be converted.

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_BIT	Data is 0 or 1.	Data	1 [c]	N/A
	Data is greater than 0, less than 2, and not equal to 1.	Truncated data	1 [c]	01004
	Data is less than 0 or greater than or equal to 2.	Untouched	Untouched	22003
	Data is not a *numeric-literal.*	Untouched	Untouched	22005

[c] This is the size of the corresponding C data type.

♦

IUS

SQL-to-C: Bit

The bit INFORMIX-CLI SQL data type is SQL_BIT. Only Universal Server supports SQL_BIT. The following table shows the INFORMIX-CLI C data types to which bit SQL data can be converted.

fCType	Test	rgbValue	pcbValue	SQLSTATE5
SQL_C_CHAR	*cbValueMax* > 1.	Data	1	N/A
	cbValueMax ≤ 1.	Untouched	Untouched	22003
SQL_C_DOUBLE SQL_C_FLOAT SQL_C_LONG SQL_C_SHORT SQL_C_SLONG SQL_C_SSHORT SQL_C_STINYINT SQL_C_TINYINT SQL_C_ULONG SQL_C_USHORT SQL_C_UTINYINT	None. [b]	Data	Size of the C data type	N/A
SQL_C_BIT	None. [b]	Data	1 [c]	N/A
SQL_C_BINARY	*cbValueMax* ≥ 1.	Data	1	N/A
	cbValueMax < 1.	Untouched	Untouched	22003

[b] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[c] This is the size of the corresponding C data type.

When bit SQL data is converted to character C data, the possible values are 0 and 1. ♦

SQL-to-C: Binary

The binary INFORMIX-CLI SQL data type is SQL_LONGVARBINARY. The following table shows the INFORMIX-CLI C data types to which binary SQL data can be converted.

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	(Length of data) * 2 < *cbValueMax*.	Data	Length of data	N/A
	(Length of data) * 2 ≥ *cbValueMax*.	Truncated data	Length of data	01004
SQL_C_BINARY	Length of data ≤ *cbValueMax*.	Data	Length of data	N/A
	Length of data > *cbValueMax*.	Truncated data	Length of data	01004

When binary SQL data is converted to character C data, each byte (8 bits) of source data is represented as two ASCII characters. These characters are the ASCII character representation of the number in its hexadecimal form. For example, a binary 00000001 is converted to "01" and a binary 11111111 is converted to "FF."

The driver always converts individual bytes to pairs of hexadecimal digits and terminates the character string with a null byte. Because of this conversion, if *cbValueMax* is even and is less than the length of the converted data, the last byte of the *rgbValue* buffer is not used. (The converted data requires an even number of bytes, the next-to-last byte is a null byte, and the last byte cannot be used.)

SQL-to-C: Date

The date INFORMIX-CLI SQL data type is SQL_DATE. The following table shows the INFORMIX-CLI C data types to which date SQL data can be converted.

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	*cbValueMax* ≥ 11.	Data	10	N/A
	cbValueMax < 11.	Untouched	Untouched	22003
SQL_C_BINARY	Length of data ≤ *cbValueMax*.	Data	Length of data	N/A
	Length of data > *cbValueMax*.	Untouched	Untouched	22003
SQL_C_DATE	None. [a]	Data	6 [c]	N/A
SQL_C_TIMESTAMP	None. [a]	Data [b]	16 [c]	N/A

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] The time fields of the time stamp structure are set to zero.

[c] This is the size of the corresponding C data type.

When date SQL data is converted to character C data, the resulting string is in the “yyyy-mm-dd” format.

SQL-to-C: Time Stamp

The time stamp INFORMIX-CLI SQL data type is SQL_TIMESTAMP. The following table shows the INFORMIX-CLI C data types to which time stamp SQL data can be converted.

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	*cbValueMax* > Display size.	Data	Length of data	N/A
	20 ≤ *cbValueMax* ≤ Display size.	Truncated data [b]	Length of data	01004
	cbValueMax < 20.	Untouched	Untouched	22003
SQL_C_BINARY	Length of data ≤ *cbValueMax*.	Data	Length of data	N/A
	Length of data > *cbValueMax*.	Untouched	Untouched	22003
SQL_C_DATE	Time portion of time stamp is zero. [a]	Data	6 [f]	N/A
	Time portion of time stamp is nonzero. [a]	Truncated data [c]	6 [f]	01004
SQL_C_TIMESTAMP	Fractional seconds portion of time stamp is not truncated. [a]	Data [e]	16 [f]	N/A
	Fractional seconds portion of time stamp is truncated. [a]	Truncated data [e]	16 [f]	01004

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] The fractional seconds of the time stamp are truncated.

[c] The time portion of the time stamp is truncated.

[d] The date portion of the time stamp is ignored.

[e] The fractional seconds portion of the time stamp is truncated.

[f] This is the size of the corresponding C data type.

When time stamp SQL data is converted to character C data, the resulting string is in the “yyyy-mm-dd hh:mm:ss[.f...]” format, where up to nine digits can be used for fractional seconds. Except for the decimal point and fractional seconds, the entire format must be used, regardless of the precision of the time stamp SQL data type.

SQL-to-C Data Conversion Examples

The following table illustrates how the driver converts SQL data to C data.

SQL Data Type	SQL Data Value	C Data Type	cbValueMax	rgbValue	SQLSTATE
SQL_CHAR	tigers	SQL_C_CHAR	7	tigers\0 [a]	N/A
SQL_CHAR	tigers	SQL_C_CHAR	6	tiger\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0 [a]	N/A
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	4	—	22003
SQL_DECIMAL	1234.56	SQL_C_FLOAT	ignored	1234.56	N/A
SQL_DECIMAL	1234.56	SQL_C_SSHORT	ignored	1234	01004
SQL_DECIMAL	1234.56	SQL_C_STINYINT	ignored	—	22003
SQL_DOUBLE	1.2345678	SQL_C_DOUBLE	ignored	1.2345678	N/A
SQL_DOUBLE	1.2345678	SQL_C_FLOAT	ignored	1.234567	N/A
SQL_DOUBLE	1.2345678	SQL_C_STINYINT	ignored	1	N/A
SQL_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0 [a]	N/A
SQL_DATE	1992-12-31	SQL_C_CHAR	10	—	22003
SQL_DATE	1992-12-31	SQL_C_TIMESTAMP	ignored	1992,12,31, 0,0,0,0 [b]	N/A
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 [a]	N/A
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 [a]	01004
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	—	22003

[a] "\0" represents a null-termination byte. The driver always null-terminates SQL_C_CHAR data.

[b] The numbers in this list are the numbers stored in the fields of the TIMESTAMP_STRUCT structure.

Converting Data from C to SQL

When an application calls **SQLExecute** or **SQLExecDirect**, the driver retrieves the data for any parameters that are bound with **SQLBindParameter** from storage locations in the application. For data-at-execution parameters, the application sends the parameter data with **SQLPutData**. If necessary, the driver converts the data from the data type specified by the *fCType* argument in **SQLBindParameter** to the data type specified by the *fSqlType* argument in **SQLBindParameter**. Finally, the driver sends the data to the data source.

The tables in the following sections describe how the driver or data source converts data sent to the data source; drivers are required to support conversions from all INFORMIX-CLI C data types to the INFORMIX-CLI SQL data types that they support. For a given INFORMIX-CLI C data type, the first column of the table lists the legal input values of the *fSqlType* argument in **SQLBindParameter**. The second column lists the outcomes of a test that the driver performs to determine whether it can convert the data. The third column lists the SQLSTATE that is returned for each outcome by **SQLExecDirect**, **SQLExecute**, or **SQLPutData**. Data is sent to the data source only if SQL_SUCCESS is returned.

If the *fSqlType* argument in **SQLBindParameter** contains a value for an INFORMIX-CLI SQL data type that is not shown in the table for a given C data type, then **SQLBindParameter** returns SQLSTATE 07006 (Restricted data type attribute violation). If the *fSqlType* argument contains a driver-specific value and the driver does not support the conversion from the specific INFORMIX-CLI C data type to the driver-specific SQL data type, then **SQLBindParameter** returns SQLSTATE S1C00 (Driver not capable).

If the *rgbValue* and *pcbValue* arguments specified in **SQLBindParameter** are both null pointers, then that function returns SQLSTATE S1009 (Invalid argument value). Although it is not shown in the tables, an application sets the value pointed to by the *pcbValue* argument of **SQLBindParameter** or the value of the *cbValue* argument to SQL_NULL_DATA to specify a NULL SQL data value. The application sets these values to SQL_NTS to specify that the value in *rgbValue* is a null-terminated string.

The following terms are used in the tables:

- *Length of data* is the number of bytes of SQL data that are available to send to the data source, regardless of whether the data is truncated before it goes to the data source. For string data, this does not include the null-termination byte.
- *Column length* and *display size* are defined for each SQL data type in "Precision, Scale, Length, and Display Size" on page B-5.
- *Number of digits* is the number of characters that represent a number, including the minus sign, decimal point, and exponent (if needed).
- Words in *italics* represent elements of the INFORMIX-CLI SQL syntax.

C-to-SQL: Character

The character INFORMIX-CLI C data type is SQL_C_CHAR. The following table shows the INFORMIX-CLI SQL data types to which C character data can be converted.

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_LONGVARCHAR SQL_VARCHAR	Length of data ≤ Column length.	N/A
	Length of data > Column length.	01004
SQL_DECIMAL SQL_INTEGER SQL_SMALLINT	Data converted without truncation.	N/A
	Data converted with truncation of fractional digits.	01004
	Conversion of data would result in loss of whole (as opposed to fractional) digits.	22003
	Data value is not a *numeric-literal*.	22005
SQL_DOUBLE SQL_REAL	Data is within the range of the data type to which the number is being converted.	N/A
	Data is outside the range of the data type to which the number is being converted.	22003
	Data value is not a *numeric-literal*.	22005

(1 of 2)

fSqlType	Test	SQLSTATE
SQL_LONGVARBINARY	(Length of data) / 2 ≤ Column length.	N/A
	(Length of data) / 2 > Column length.	01004
	Data value is not a hexadecimal value.	22005
SQL_DATE	Data value is a valid INFORMIX-CLI-*date-literal.*	N/A
	Data value is a valid INFORMIX-CLI-*timestamp-literal*; time portion is zero.	N/A
	Data value is a valid INFORMIX-CLI-*timestamp-literal*; time portion is non-zero.[a]	01004
	Data value is not a valid INFORMIX-CLI-*date-literal* or INFORMIX-CLI-*timestamp-literal.*	22008
SQL_TIMESTAMP	Data value is a valid INFORMIX-CLI-*timestamp-literal*; fractional seconds portion not truncated.	N/A
	Data value is a valid INFORMIX-CLI-*timestamp-literal*; fractional seconds portion truncated.	01004
	Data value is a valid INFORMIX-CLI-*date-literal.* [d]	N/A
	Data value is a valid INFORMIX-CLI-*time-literal.* [e]	N/A
	Data value is not a valid INFORMIX-CLI-*date-literal,* INFORMIX-CLI-*time-literal,* or INFORMIX-CLI-*timestamp-literal.*	22008

[a] The time portion of the time stamp is truncated.
[b] The date portion of the time stamp is ignored.
[c] The fractional seconds portion of the time stamp is truncated.
[d] The time portion of the time stamp is set to zero.
[e] The date portion of the time stamp is set to the current date.

(2 of 2)

When character C data is converted to date or time stamp SQL data, leading and trailing blanks are ignored.

When character C data is converted to numeric, date, or time stamp SQL data, leading and trailing blanks are ignored.

When character C data is converted to binary SQL data, each 2 bytes of character data are converted to a single byte (8 bits) of binary data. Each 2 bytes of character data represent a number in hexadecimal form. For example, "01" is converted to a binary 00000001 and "FF" is converted to a binary 11111111.

The driver always converts pairs of hexadecimal digits to individual bytes and ignores the null-termination byte. Because of this conversion, if the length of the character string is odd, the last byte of the string (excluding the null termination byte, if any) is not converted.

IUS

Additional C-to-SQL Character Data Conversion for Universal Server

The following table shows the additional INFORMIX-CLI SQL data types for Universal Server to which character C data can be converted.

fSqlType	Test	SQLSTATE
SQL_BIGINT	Data converted without truncation.	N/A
	Data converted with truncation of fractional digits.	01004
	Conversion of data would result in loss of whole (as opposed to fractional) digits.	22003
	Data value is not a *numeric-literal*.	22005
SQL_BIT	Data is 0 or 1.	N/A
	Data is greater than 0, less than 2, and not equal to 1.	01004
	Data is less than 0 or greater than or equal to 2.	22003
	Data is not a *numeric-literal*.	22005

♦

C-to-SQL: Numeric

The numeric INFORMIX-CLI C data types are:

- SQL_C_DOUBLE
- SQL_C_FLOAT
- SQL_C_LONG
- SQL_C_SHORT
- SQL_C_SLONG
- SQL_C_SSHORT
- SQL_C_STINYINT
- SQL_C_TINYINT
- SQL_C_ULONG
- SQL_C_USHORT
- SQL_C_UTINYINT

The following table shows the INFORMIX-CLI SQL data types to which numeric C data can be converted.

fSqlType	**Test**	**SQLSTATE**
SQL_CHAR SQL_LONGVARCHAR SQL_VARCHAR	Number of digits ≤ Column length.	N/A
	Number of whole (as opposed to fractional) digits ≤ Column length.	01004
	Number of whole (as opposed to fractional) digits > Column length.	22003
SQL_DECIMAL SQL_INTEGER SQL_SMALLINT	Data converted without truncation.	N/A
	Data converted with truncation of fractional digits.	01004
	Conversion of data would result in loss of whole (as opposed to fractional) digits.	22003
SQL_DOUBLE SQL_REAL	Data is within the range of the data type to which the number is being converted.	N/A
	Data is outside the range of the data type to which the number is being converted.	22003

The value pointed to by the *pcbValue* argument of **SQLBindParameter** and the value of the *cbValue* argument of **SQLPutData** are ignored when data is converted from the numeric C data types. The driver assumes that the size of *rgbValue* is the size of the numeric C data type.

IUS

Additional C-to-SQL Numeric Data Conversion for Universal Server

The following table shows the additional INFORMIX-CLI SQL data types for Universal Server to which numeric C data can be converted.

fSqlType	Test	SQLSTATE
SQL_BIGINT	Data converted without truncation.	N/A
	Data converted with truncation of fractional digits.	01004
	Conversion of data would result in loss of whole (as opposed to fractional) digits.	22003
SQL_BIT	Data is 0 or 1.	N/A
	Data is greater than 0, less than 2, and not equal to 1.	01004
	Data is less than 0 or greater than or equal to 2.	22003

♦

IUS

C-to-SQL: Bit

The bit INFORMIX-CLI C data type is SQL_C_BIT. Only Universal Server supports SQL_C_BIT. The following table shows the INFORMIX-CLI SQL data types to which bit C data can be converted.

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_LONGVARCHAR SQL_VARCHAR	None.	N/A
SQL_BIGINT SQL_DECIMAL SQL_DOUBLE SQL_INTEGER SQL_REAL SQL_SMALLINT	None.	N/A
SQL_BIT	None.	N/A

The value pointed to by the *pcbValue* argument of **SQLBindParameter** and the value of the *cbValue* argument of **SQLPutData** are ignored when data is converted from the bit C data type. The driver assumes that the size of *rgbValue* is the size of the bit C data type. ♦

C-to-SQL: Binary

The binary INFORMIX-CLI C data type is SQL_C_BINARY. The following table shows the INFORMIX-CLI SQL data types to which binary C data can be converted.

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_LONGVARCHAR SQL_VARCHAR	Length of data ≤ Column length. Length of data > Column length.	N/A 01004
SQL_DECIMAL SQL_DOUBLE SQL_INTEGER SQL_REAL SQL_SMALLINT	Length of data = SQL data length. [a] Length of data ≠ SQL data length. [a]	N/A 22003
SQL_LONGVARBINARY	Length of data ≤ Column length. Length of data > Column length.	N/A 01004
SQL_DATE SQL_TIMESTAMP	Length of data = SQL data length. [a] Length of data ≠ SQL data length. [a]	N/A 22003

[a] The SQL data length is the number of bytes needed to store the data on the data source. This may be different than the column length, as defined in "Precision, Scale, Length, and Display Size" on page B-5.

IUS

Additional C-to-SQL Binary Data Conversion for Universal Server

The following table shows the additional INFORMIX-CLI SQL data types for Universal Server to which binary C data can be converted.

fSqlType	Test	SQLSTATE
SQL_BIGINT	Length of data = SQL data length. [a] Length of data ≠ SQL data length. [a]	N/A 22003
SQL_BIT	Length of data = SQL data length. [a] Length of data ≠ SQL data length. [a]	N/A 22003

♦

C-to-SQL: Date

The date INFORMIX-CLI C data type is SQL_C_DATE. The following table shows the INFORMIX-CLI SQL data types to which date C data can be converted.

fSqlType	**Test**	**SQLSTATE**
SQL_CHAR SQL_LONGVARCHAR SQL_VARCHAR	Column length $\geq$ 10.	N/A
	Column length < 10.	22003
	Data value is not a valid date.	22008
SQL_DATE	Data value is a valid date.	N/A
	Data value is not a valid date.	22008
SQL_TIMESTAMP	Data value is a valid date. [a]	N/A
	Data value is not a valid date.	22008

[a] The time portion of the time stamp is set to zero.

For information about what values are valid in a SQL_C_DATE structure, see "Date and Time Stamp C Data Types" on page B-11.

When date C data is converted to character SQL data, the resulting character data is in the "yyyy-mm-dd" format.

The value pointed to by the *pcbValue* argument of **SQLBindParameter** and the value of the *cbValue* argument of **SQLPutData** are ignored when data is converted from the date C data type. The driver assumes that the size of *rgbValue* is the size of the date C data type.

C-to-SQL: Time Stamp

The time stamp INFORMIX-CLI C data type is SQL_C_TIMESTAMP. The following table shows the INFORMIX-CLI SQL data types to which time stamp C data can be converted.

fSqlType	Test	SQLSTATE
SQL_CHAR SQL_LONGVARCHAR SQL_VARCHAR	Column length ≥ Display size.	N/A
	19 ≤ Column length < Display size. [a]	01004
	Column length < 19.	22003
	Data value is not a valid date.	22008
SQL_DATE	Time fields are zero.	N/A
	Time fields are non-zero. [b]	01004
	Data value does not contain a valid date.	22008
SQL_TIMESTAMP	Fractional seconds fields are not truncated.	N/A
	Fractional seconds fields are truncated. [d]	01004
	Data value is not a valid time stamp.	22008

[a] The fractional seconds of the time stamp are truncated.
[b] The time fields of the time stamp structure are truncated.
[c] The date fields of the time stamp structure are ignored.
[d] The fractional seconds fields of the time stamp structure are truncated.

For information about what values are valid in a SQL_C_TIMESTAMP structure, see "Date and Time Stamp C Data Types" on page B-11.

When time stamp C data is converted to character SQL data, the resulting character data is in the "yyyy-mm-dd hh:mm:ss[.f...]" format.

The value pointed to by the *pcbValue* argument of **SQLBindParameter** and the value of the *cbValue* argument of **SQLPutData** are ignored when data is converted from the time stamp C data type. The driver assumes that the size of *rgbValue* is the size of the time stamp C data type.

C-to-SQL Data Conversion Examples

The following table illustrates how the driver converts C data to SQL data.

C Data Type	C Data Value	SQL Data Type	Column length	SQL Data Value	SQLSTATE
SQL_C_CHAR	tigers\0 [a]	SQL_CHAR	6	tigers	N/A
SQL_C_CHAR	tigers\0 [a]	SQL_CHAR	5	tiger	01004
SQL_C_CHAR	1234.56\0 [a]	SQL_DECIMAL	8 [b]	1234.56	N/A
SQL_C_CHAR	1234.56\0 [a]	SQL_DECIMAL	7 [b]	1234.5	01004
SQL_C_CHAR	1234.56\0 [a]	SQL_DECIMAL	4	—	22003
SQL_C_FLOAT	1234.56	SQL_FLOAT	not applicable	1234.56	N/A
SQL_C_FLOAT	1234.56	SQL_INTEGER	not applicable	1234	01004
SQL_C_FLOAT	1234.56	SQL_TINYINT	not applicable	—	22003
SQL_C_DATE	1992,12,31 [c]	SQL_CHAR	10	1992-12-31	N/A
SQL_C_DATE	1992,12,31 [c]	SQL_CHAR	9	—	22003
SQL_C_DATE	1992,12,31 [c]	SQL_TIMESTAMP	not applicable	1992-12-31 00:00:00.0	N/A
SQL_C_TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	22	1992-12-31 23:45:55.12	N/A
SQL_C_TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	21	1992-12-31 23:45:55.1	01004
SQL_C_TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	18	—	22003

[a] "\0" represents a null-termination byte. The null-termination byte is required only if the length of the data is SQL_NTS.

[b] In addition to bytes for numbers, one byte is required for a sign and another for the decimal point.

[c] The numbers in this list are the numbers stored in the fields of the DATE_STRUCT structure.

[d] The numbers in this list the numbers stored in the fields of the TIMESTAMP_STRUCT structure.

Comparison of INFORMIX-CLI and Embedded SQL

This appendix compares INFORMIX-CLI and embedded SQL.

INFORMIX-CLI to Embedded SQL

The following table compares INFORMIX-CLI functions (core ODBC) with embedded SQL statements. This comparison is based on the X/Open and SQL Access Group SQL CAE specification (1992).

INFORMIX-CLI uses a parameter marker in place of a host variable, where a host variable occurs in embedded SQL.

The SQL language is based on the X/Open and SQL Access Group SQL CAE specification (1992).

INFORMIX-CLI Function	Embedded SQL Statement	Comments
SQLAllocEnv	none	Driver manager and driver memory allocation.
SQLAllocConnect	none	Driver manager and driver memory allocation.
SQLConnect	CONNECT	Association management.
SQLAllocStmt	none	Driver manager and driver memory allocation.
SQLPrepare	PREPARE	The prepared SQL string can contain any of the valid preparable functions that the X/Open specification defines, including ALTER, CREATE, *cursor-specification*, searched DELETE, dynamic SQL positioned DELETE, DROP, GRANT, INSERT, REVOKE, searched UPDATE, or dynamic SQL positioned UPDATE.
SQLBindParameter	SET DESCRIPTOR	Dynamic SQL ALLOCATE DESCRIPTOR and dynamic SQL SET DESCRIPTOR. ALLOCATE DESCRIPTOR would normally be issued on the first call to **SQLBindParameter** for an *hstmt*. Alternatively, ALLOCATE DESCRIPTOR can be called during **SQLAllocStmt**, although this call would not be needed by SQL statements that do not contain embedded parameters. The driver generates the descriptor name.
SQLSetCursorName	none	The specified cursor name is used in the DECLARE CURSOR statement that **SQLExecute** or **SQLExecDirect** generates.

(1 of 3)

INFORMIX-CLI Function	Embedded SQL Statement	Comments
SQLGetCursorName	none	Driver cursor name management.
SQLExecute	EXECUTE or DECLARE CURSOR and OPEN CURSOR	Dynamic SQL EXECUTE. If the SQL statement requires a cursor, then a dynamic SQL DECLARE CURSOR statement and a dynamic SQL OPEN are issued at this time.
SQLExecDirect	EXECUTE IMMEDIATE or DECLARE CURSOR and OPEN CURSOR	The INFORMIX-CLI function call provides for support for a *cursor specification* and statements allowed in an EXECUTE IMMEDIATE dynamic SQL statement. In the case of a *cursor specification*, the call corresponds to static SQL DECLARE CURSOR and OPEN statements.
SQLNumResultCols	GET DESCRIPTOR	COUNT form of dynamic SQL GET DESCRIPTOR.
SQLColAttributes	GET DESCRIPTOR	COUNT form of dynamic SQL GET DESCRIPTOR or VALUE form of dynamic SQL GET DESCRIPTOR with *field-name* in {NAME, TYPE, LENGTH, PRECISION, SCALE, NULLABLE}.
SQLDescribeCol	GET DESCRIPTOR	VALUE form of dynamic SQL GET DESCRIPTOR with *field-name* in {NAME, TYPE, LENGTH, PRECISION, SCALE, NULLABLE}.
SQLBindCol	none	This function establishes output buffers that correspond in usage to host variables for static SQL FETCH, and to an SQL DESCRIPTOR for dynamic SQL FETCH *cursor* USING SQL DESCRIPTOR *descriptor.*

(2 of 3)

INFORMIX-CLI Function	Embedded SQL Statement	Comments
SQLFetch	FETCH	Static or dynamic SQL FETCH. If the call is a dynamic SQL FETCH, then the VALUE form of GET DESCRIPTOR is used, with *field-name* in {DATA, INDICATOR}. DATA and INDICATOR values are placed in output buffers specified in **SQLBindCol**.
SQLRowCount	GET DIAGNOSTICS	Requested field ROW_COUNT.
SQLFreeStmt (SQL_CLOSE option)	CLOSE	Dynamic SQL CLOSE.
SQLFreeStmt (SQL_DROP option)	none	Driver manager and driver memory deallocation.
SQLTransact	COMMIT WORK or COMMIT ROLLBACK	None.
SQLDisconnect	DISCONNECT	Association management.
SQLFreeConnect	none	Driver manager and driver memory deallocation.
SQLFreeEnv	none	Driver manager and driver memory deallocation.
SQLCancel	none	None.
SQLError	GET DIAGNOSTICS	GET DIAGNOSTICS retrieves information from the SQL diagnostics area that pertains to the most recently executed SQL statement. This information can be retrieved following execution and preceding the deallocation of the statement.

(3 of 3)

Embedded SQL to INFORMIX-CLI

The following tables list the relationship between the X/Open embedded SQL language and corresponding INFORMIX-CLI functions. The section number that appears in the first column of each table refers to the section of the X/Open and SQL Access Group SQL CAE specification (1992).

Declarative Statements

The following table lists declarative statements.

Section	SQL Statement	INFORMIX-CLI Function	Comments
4.3.1	Static SQL DECLARE CURSOR	none	Issued implicitly by the driver if a *cursor specification* is passed to **SQLExecDirect**.
4.3.2	Dynamic SQL DECLARE CURSOR	none	The driver automatically generates the cursor. To set a name for the cursor, use **SQLSetCursorName**. To retrieve a cursor name, use **SQLGetCursorName**.

Data Definition Statements

The following table lists data definition statements.

Section	SQL Statement	INFORMIX-CLI Function	Comments
5.1.2 5.1.3 5.1.4 5.1.5 5.1.6 5.1.7 5.1.8 5.1.9	ALTER TABLE CREATE INDEX CREATE TABLE CREATE VIEW DROP INDEX DROP TABLE DROP VIEW GRANT REVOKE	**SQLPrepare**, **SQLExecute**, or **SQLExecDirect**	None.

Data Manipulation Statements

The following table lists data manipulation statements.

Section	SQL Statement	INFORMIX-CLI Function	Comments
5.2.1	CLOSE	**SQLFreeStmt** (SQL_CLOSE option)	None.
5.2.2	Positioned DELETE	**SQLExecDirect**(..., "DELETE FROM *table-name* WHERE CURRENT OF *cursor-name*")	Driver-generated *cursor-name* can be obtained by calling **SQLGetCursorName**.
5.2.3	Searched DELETE	**SQLExecDirect**(..., "DELETE FROM *table-name* WHERE *search-condition*")	None.
5.2.4	FETCH	**SQLFetch**	None.
5.2.5	INSERT	**SQLExecDirect** (...,"INSERT INTO *table-name*...")	Can also be invoked by **SQLPrepare** and **SQLExecute**.
5.2.6	OPEN	none	When a SELECT statement is specified, a cursor is opened implicitly by **SQLExecute** or **SQLExecDirect**.
5.2.7	SELECT...INTO	none	Not supported.
5.2.8	Positioned UPDATE	**SQLExecDirect**(..., "UPDATE *table-name* SET *column-identifier* = *expression*...WHERE CURRENT OF *cursor-name*")	Driver-generated *cursor-name* can be obtained by calling **SQLGetCursorName**.
5.2.9	Searched UPDATE	**SQLExecDirect**(..., "UPDATE *table-name* SET *column-identifier* = *expression*...WHERE *search-condition*")	None.

Dynamic SQL Statements

The following table lists dynamic SQL statements.

Section	SQL Statement	INFORMIX-CLI Function	Comments
5.3 (see 5.2.1)	Dynamic SQL CLOSE	**SQLFreeStmt** (SQL_CLOSE option)	None.
5.3 (see 5.2.2)	Dynamic SQL Positioned DELETE	**SQLExecDirect**(..., "DELETE FROM *table-name* WHERE CURRENT OF *cursor-name*")	Can also be invoked by **SQLPrepare** and **SQLExecute**.
5.3 (see 5.2.8)	Dynamic SQL Positioned UPDATE	**SQLExecDirect**(..., "UPDATE *table-name* SET *column-identifier* = *expression*...WHERE CURRENT OF *cursor-name*")	Can also be invoked by **SQLPrepare** and **SQLExecute**.
5.3.3	ALLOCATE DESCRIPTOR	None	Descriptor information is implicitly allocated and attached to the *hstmt* by the driver. Allocation occurs at either the first call to **SQLBindParameter** or at **SQLExecute** or **SQLExecDirect** time.
5.3.4	DEALLOCATE DESCRIPTOR	**SQLFreeStmt** (SQL_DROP option)	None.
5.3.5	DESCRIBE	none	None.
5.3.6	EXECUTE	**SQLExecute**	None.
5.3.7	EXECUTE IMMEDIATE	**SQLExecDirect**	None.
5.3.8	Dynamic SQL FETCH	**SQLFetch**	None.

(1 of 2)

Section	SQL Statement	INFORMIX-CLI Function	Comments
5.3.9	GET DESCRIPTOR	**SQLNumResultCols** **SQLDescribeCol** **SQLColAttributes**	COUNT FORM. VALUE form with *field-name* in {NAME, TYPE, LENGTH, PRECISION, SCALE, NULLABLE}.
5.3.10	Dynamic SQL OPEN	**SQLExecute**	None.
5.3.11	PREPARE	**SQLPrepare**	None.
5.3.12	SET DESCRIPTOR	**SQLBindParameter**	**SQLBindParameter** is associated with only one *hstmt* where a descriptor is applied to any number of statements with USING SQL DESCRIPTOR.

(2 of 2)

Transaction Control Statements

The following table lists transaction control statements.

Section	SQL Statement	INFORMIX-CLI Function	Comments
5.4.1	COMMIT WORK	**SQLTransact** (SQL_COMMIT option)	None.
5.4.2	ROLLBACK WORK	**SQLTransact** (SQL_ROLLBACK option)	None.

Association Management Statements

The following table lists association management statements.

Section	SQL Statement	INFORMIX-CLI Function	Comments
5.5.1	CONNECT	**SQLConnect**	None.
5.5.2	DISCONNECT	**SQLDisconnect**	INFORMIX-CLI does not support DISCONNECT ALL.
5.5.3	SET CONNECTION	None	The SQL Access Group (SAG) Call Level Interface allows multiple simultaneous connections to be established, but only one connection can be active at one time. SAG-compliant drivers track which connection is active and automatically switch to a different connection if a different connection handle is specified. However, the active connection must be in a state that allows the connection context to be switched; in other words, a transaction must not be in progress on the current connection. Drivers that are not SAG-compliant are not required to support this behavior. That is, drivers that are not SAG- compliant are not required to return an error if the driver, and its associated data source can simultaneously support multiple active connections.

Diagnostic Statement

The following table lists the GET DIAGNOSTIC statement.

Section	SQL Statement	INFORMIX-CLI Function	Comments
5.6.1	GET DIAGNOSTICS	**SQLError** **SQLRowCount**	For **SQLError**, the following fields from the diagnostics area are available: RETURNED_SQLSTATE, MESSAGE_TEXT, and MESSAGE_LENGTH. For **SQLRowCount**, the ROW_COUNT field is available.

Index